MASTER EN SCIENCES INFORMATIQUES

Mesures de performance de systèmes-experts

Delaisse, Luc

Award date:
1986

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix. Namur.

Mesures de performance de systèmes-experts

DELAISSE Luc

Thèse présentée en vue de l'obtention du grade de licencié et maître en Informatique. Septembre 1986.

Sous la direction de Monsieur Jean Ramaekers.

Je remercie vivement :

Monsieur Marcel CLANTIN, pour la rigueur et la patience avec lesquelles il a relu le présent travail, ainsi que pour les conseils précieux qu'il m'a donnés,

Monsieur Yves DEVILLE, pour la clarté de ses explications et de ses recommandations,

Monsieur Jean-Jacques GUILLEMAUX, pour son accueil et ses conseils,

Monsieur Jean RAMAEKERS, pour ses remarques toujours pertinentes et constructives,

Monsieur Pascal VANHENTENRYCK, pour ses avis judicieux et l'abondante documentation qu'il m'a procurée,

Monsieur Axel VAN LAMSWEERDE, pour m'avoir initié à l'intelligence artificielle.

TABLE DES MATIERES

RESUME

Ce travail a pour objet de trouver un certain nombre d'idées pour mesurer et comparer des systèmes-experts. Il s'agit à la fois d'un travail de recherche et de compilation d'études récentes en la matière.

Après une introduction aux systèmes-experts et à leurs aspects performances, nous analyserons en détail les systèmes KOOL et Prolog.

Nous terminerons par une discussion sur OPS5 et sur les formalismes de représentation des connaissances.

CHAPITRE I : Introduction

Rarement l'informatique aura vu une de ses disciplines ou un de ses sous-produits passer si rapidement de l'étape de la recherche à celle du marketing. Car aujourd'hui, l'intelligence artificielle, c'est un message que l'on assène, c'est une profession de foi que l'on répète à l'envi.

I.1. Qu'est-ce qu'un système-expert ?

I.1.1. Historique

Les systèmes-experts sont le résultat de recherches récentes en Intelligence Artificielle (IA).

L'IA est cette partie de l'informatique qui se préoccupe de la prise en charge par l'ordinateur de tâches intellectuelles.

Les premières recherches en IA furent très ambitieuses : elles concernaient des techniques de raisonnement simples et puissantes pouvant s'appliquer à des problèmes quelconques. Un exemple frappant du résultat de telles recherches est le programme GPS (General Problem Solver) de Newell et Simon.

L'IA envisageait donc de comprendre la manière dont un organisme traite « une étendue de problèmes ... coextensive à celle accessible à l'esprit humain ».

Cependant, lorsqu'il s'agit d'appliquer ces méthodes à des problèmes du monde réel, elles se révèlent insuffisantes et beaucoup moins performantes qu'un spécialiste du domaine.

Cela est principalement dû, comme l'a fait remarquer Feigenbaum, au fait qu'un spécialiste possède une masse considérable de connaissances sur son domaine et qu'il résout rarement un problème à partir de principes de base. L'IA reconnaît aujourd'hui que la connaissance est aussi importante que le raisonnement. La puissance réside dans la connaissance spécifique relative au domaine du problème. Les systèmes les plus puissants sont ceux qui possèdent le plus de connaissances.

C'est ainsi qu'ont commencé à se développer toute une famille de systèmes basés sur la connaissance (Knowledge-Based Systems) appelés systèmes-experts; ce sont des programmes qui exploitent la connaissance concernant un domaine bien défini pour résoudre dans ce domaine des problèmes difficiles, leur but étant de parvenir à des performances égales à celles des meilleurs spécialistes.

D'autre part, les systèmes-experts sont avant tout conçus pour l'homme : ils doivent donc constituer des outils facilement compréhensibles et aisément modifiables.

Les exemples de systèmes-experts sont nombreux; on peut citer parmi les plus célèbres : DENDRAL dans le domaine de la chimie organique, MYCIN dans celui des maladies bactériennes du sang, PROSPECTOR en géologie, MECHO en mécanique, R1 en configuration de systèmes, ...

I.1.2. Définition d'un système-expert

Un **système-expert** est un système de représentation et d'utilisation de connaissances caractérisé d'une part par ses objectifs et d'autre part par sa structure d'implémentation [Rousset 83].

a) les objectifs :

L'objectif principal est de modéliser le raisonnement d'un expert.

Un système-expert doit être capable :

- d'utiliser de nombreuses connaissances sur un domaine précis. Ces connaissances correspondent à l'expérience qu'ont pu acquérir des spécialistes de ce domaine.
- de raisonner avec efficacité et rapidité sur ces connaissances.
- dans certains domaines, de faire un raisonnement approximatif sur des connaissances pouvant être imprécises, incertaines ou incomplètes. Un exemple typique d'un tel domaine est la médecine : les premiers systèmes-experts opérationnels ont d'ailleurs été réalisés pour modéliser le diagnostic médical sur certains types de maladie.
- de justifier son comportement.

b) la structure d'implémentation :

Elle est caractérisée par la *séparation* entre :

- la connaissance propre au domaine, donnée en vrac, de façon déclarative.
- le contrôle sur cette connaissance, c'est-à-dire les algorithmes de résolution utilisés.

Il existe trois grands formalismes de représentation des connaissances : les règles de production, les objets structurés et la logique des prédicats.

I.1.2.1. Les règles de production

Un **système de production** consiste en un ensemble de règles et un interpréteur de ces règles qui décide *quelle* règle appliquer et *quand* l'appliquer.

Les **règles** sont constituées de paires prémisses-actions du type

si P_1 & P_2 & ... & P_n *alors* A_1 & ... & A_m.

Une telle règle peut être interprétée comme : *si* P_1 *et* P_2 *et ... et* P_n *sont vrais alors exécuter les actions* A_1 *et ... et* A_m.

Par exemple :

si X (est mammifère) (est carnivore) (est de couleur fauve) (a des rayures noires) alors c'est un tigre.

I.1.2.2. Les objets structurés

Les **objets structurés** font référence aux schémas de représentation basés sur la théorie des graphes ou aux structures d'enregistrement avec des *slots* (en français encoches) et des *fillers* (en français remplisseurs). Dans ce dernier cas, on parlera de système de *frames* (cadres en français).

Un frame pourrait par exemple se présenter comme suit :

[Frame N 137 : Jean
âge : 32
taille : 1,78
poids : 72
]

Un système de frames inclut des procédures de création, manipulation et destruction de cadres. L'utilisateur peut ajouter des informations à un enregistrement et modifier les enregistrements qui existent déjà. Les slots peuvent aussi contenir des procédures écrites dans un langage donné, procédures qui pourront ensuite être exécutées dans certaines circonstances.

I.1.2.3. La logique des prédicats

La **logique des prédicats** permet, grâce à un formalisme particulier, de réaliser l'analyse de propositions et de raisonner avec des expressions quantifiées [Jackson 85]. Dans le calcul des propositions, on effectue des raisonnements logiques sur des propositions dans lesquelles des variables, telles *P* et *Q*, peuvent représenter des expressions comme *"Socrate est un homme"* et *"Socrate est mortel"*.

La **programmation logique** n'est qu'une vue dérivée par Kowalski [Kowalski 79] de la logique des prédicats du 1er ordre. On trouvera au chapitre III.1. une description plus complète de la programmation logique.

I.1.3. Structure d'un système-expert

Un système-expert est classiquement constitué de trois modules indépendants et coopérant entre eux :

- la base de connaissances,
- la base de faits,
- le moteur d'inférences.

La **base de connaissances** contient l'ensemble des informations spécifiques au domaine d'expertise. Elle est écrite dans un langage de représentation des connaissances où l'expert peut - ou devrait pouvoir - définir son propre vocabulaire. A l'inverse de ce qui se passe dans un programme classique, les informations sont entrées "en vrac" et l'ordre n'influe pas sur les résultats. Ces connaissances sont souvent stockées sous forme de règles.

Puisque ces connaissances représentent l'expertise du domaine considéré, elles peuvent correspondre à un savoir-faire, à des stratégies, à des formules, à la description du domaine ou à un mélange savamment dosé de ces notions.

Nous avons vu que plusieurs formalismes pouvaient être utilisés pour exprimer les connaissances. Dans la suite de cette introduction, nous ne tiendrons compte que des règles de production.

La **base de faits**, aussi appelée mémoire de travail, contient les données propres aux problèmes à traiter. Elle représente à tout moment la connaissance que le système a acquise au sujet du problème particulier qu'il est en train de traiter. En mémorisant tous les résultats intermédiaires, la mémoire de travail conserve une trace des raisonnements effectués; elle peut donc être utilisée, à la fois pour expliquer l'origine des informations déduites au cours d'une session, et pour décrire le comportement du système.

Le **moteur d'inférences** est le coeur du système-expert. Il contient les mécanismes de raisonnement que celui-ci utilise à un moment donné pour, à partir des données de la base de faits décrivant la connaissance que le système a du problème considéré, décider dynamiquement de l'enchaînement des règles de la base de connaissances. Deux modes de raisonnement peuvent être utilisés :

- le **chaînage-avant** ou raisonnement dirigé par les données. Il correspond à exploiter toute règle de production < *MembreGauche* --> *MembreDroit* > de la façon suivante :
  si la conjonction des conditions constituant MG est vraie, alors on en déduit que les conclusions de MD sont vraies.
  Le raisonnement se fait en utilisant les règles à partir des faits connus jusqu'à ce qu'un but ait été déduit. Lorsqu'une des règles est employée, les nouveaux faits sont ajoutés à la base de faits.
- le **chaînage-arrière** se définit par rapport à un but. Il correspond à décomposer ce but en sous-buts de la façon suivante :
  une façon de démontrer B, terme du membre droit d'une règle < MG --> MD >, *est de démontrer les prémisses constituant* MG *qui deviennent autant de sous-problèmes (ou sous-buts).*

L'enchaînement des règles peut alors se représenter par un arbre ET/OU construit à partir du but recherché. Un arbre ET/OU est un arbre dont les noeuds "ET" signifient que, tôt ou tard, chacun des différents littéraux d'un but devra être vérifié et les noeuds "OU" représentent l'existence d'hypothèses alternatives pour le faire [Winston 77].

- Chaînage-avant ou chaînage-arrière ?

On peut se demander quel est le type de raisonnement qu'il convient d'adopter lorsqu'on construit un système expert :

- si le nombre de conclusions, à priori possibles, est peu élevé, le chaînage-arrière est avantageux car il permet de se focaliser sur le but; pour chaque but envisagé on ne considère que les faits pouvant mener à ce but,
- cependant, cette démarche n'est pas toujours possible parce que l'on ne poursuit pas un but précis, ou parce qu'il existe un assez grand nombre de buts possibles. Le cas échéant, on adoptera d'emblée le chaînage-avant.

Un compromis intéressant entre ces deux démarches consiste à raisonner en deux étapes :

1) étape d'invocation des buts : on peut raisonner en chaînage-avant pour déterminer, à partir des données, un ensemble assez restreint de buts envisageables;
2) étape de validation des buts : on peut alors raisonner en chaînage-arrière à partir de chacun de ces buts pour tenter de les valider (ou de les invalider).

Fonctionnement du moteur d'inférences :

Tant qu'il reste des règles pertinentes, le moteur d'inférences procède en effectuant une séquence de cycles élémentaires, tous semblables. L'exécution d'un cycle de base correspond aux 3 étapes :

1. détermination des règles pertinentes (ou ensemble de conflits) par mise en correspondance (ou opération de "*pattern-matching*") des éléments de la base de faits avec la base de connaissances,

2. choix, parmi cet ensemble, de la règle à utiliser (ou résolution de conflit),
3. utilisation de cette règle (avec les mises à jour en conséquence de la base de faits).

Si le type de raisonnement est le chaînage-avant :
- Une règle est pertinente si elle a toutes ses conditions vérifiées par des éléments de la base de faits.
- L'utilisation d'une règle pertinente consiste à la déclencher, c'est-à-dire d'ajouter ses conclusions à la mémoire de travail.

Si le type de raisonnement est le chaînage-arrière :
La tâche du moteur d'inférences consiste à construire au cours des cycles un arbre ET/OU à partir du but initial et de l'ensemble de la base de connaissances.
Ainsi, à chaque cycle, on a un ensemble de buts B (correspondant aux feuilles d'un arbre ET/OU qu'on développe), B étant au premier cycle réduit à la conjonction des buts de départ.
- Une règle est pertinente si elle contient un des buts B_i en conclusion. On dira qu'il y a unification entre la règle et le but.
- L'utilisation d'une telle règle consiste à remplacer B_i par les conditions de cette règle qui deviennent autant de nouveaux buts : cela revient, dans l'arbre ET/OU, à donner comme successeur au noeud représentant B_i un noeud ET regroupant autant de feuilles que de conditions de la règle considérée.
- L'étape de choix (étape 2 du cycle de base) consiste à choisir le but courant à développer d'une part et, d'autre part, la règle à utiliser pour développer ce but.

I.1.4. Les champs d'application des systèmes-experts

Les systèmes-experts constituent une nouvelle classe d'outils informatiques capables de résoudre des problèmes sur lesquels jusqu'à présent l'informatique classique a échoué pour une des raisons suivantes :

- on connaît des algorithmes capables de résoudre le problème considéré, mais leur trop grande complexité les rend irréalisables dans la pratique : un exemple d'un tel domaine est celui de la théorie des jeux.
- on ne connaît aucun algorithme capable de résoudre le problème : c'est le cas dans de nombreux domaines où la connaissance est un savoir-faire basé sur l'expérience comme la médecine, par exemple.

Cependant, cette nouvelle technique en est encore au stade de la recherche, et des améliorations sont à envisager dans plusieurs directions selon le domaine traité :

- la puissance d'expression des langages de représentation des connaissances,
- la sophistication du raisonnement : aucun système-expert ne semble capable de raisonner par supposition,
- l'explication de son raisonnement par le système,
- l'apprentissage par le système de nouvelles connaissances, acquises lors de la résolution de problèmes antérieurs.

Depuis quelques années sont apparus sur le marché des "*systèmes-experts vides*" ou "*systèmes généraux*".

En gros, un **système général** réunit un moteur d'inférences, un langage d'expression (externe) des connaissances, des structures et conventions de représentation (interne) de ces connaissances. Un système général ne contient pas de connaissances spécifiques d'un domaine particulier d'application. Par abus de langage, les systèmes généraux sont parfois appelés des *moteurs* alors qu'ils ne sont pas que cela. Lorsqu'on complète un système général en chargeant sa base de connaissances en vue d'une application particulière on dit souvent qu'on *instantie* le système général.

I.2. Les mesures de performance des systèmes-experts

Les aspects performances des systèmes-experts ont jusqu'ici été peu analysés. Pourtant, on rencontre parfois des publicités affirmant : le système X est plus rapide que le système Y. D'accord, mais sur quels arguments cette affirmation repose-t-elle ?

Pour les langages conventionnels, il existe des **programmes étalons** écrits dans un langage portable et accompagnés de **protocoles de mesures** (par exemple le programme Fortran Whetstone).

Ces programmes étalons ne fournissent cependant que des indices de comportement. En effet, ils se "contentent" de tester quelques aspects du langage et ne prétendent en aucun cas en faire une analyse exhaustive. Ils ne permettent pas par exemple d'affirmer qu'un projet - de taille non négligeable - s'exécutera plus rapidement avec le compilateur X qu'avec le compilateur Y d'un langage donné.

Que dire alors de la comparaison, sur base de tels programmes étalons, de langages différents, si ce n'est qu'elle n'a guère de sens. Cela revient en quelque sorte à comparer des pommes et des poires. On ne peut en effet raisonnablement comparer que des langages dont les implémentations et les spécificités ont des correspondances bien définies.

De même, il importe de ne comparer que des produits développés sur une même machine. En effet, la plupart des langages font appels, dans une certaine mesure, au *hardware* qui les supporte. Tenter de comparer ainsi différentes implémentations entraînerait donc aussi une comparaison implicite des hardwares sur lesquels elles s'exécutent.

Quant aux systèmes-experts, il ne paraît pas exister de programmes étalons qui permettent de les mesurer et de les comparer.

Pourtant, la rapidité d'un compilateur ou d'un interpréteur est souvent donnée et mesurée en **inférences logiques par seconde** (**LIPS** : ***L**ogical **I**nferences **P**er **S**econd*), où une inférence logique correspond à l'accomplissement d'un cycle de résolution.

Cette mesure est intéressante, mais elle n'a de sens que si toutes les unifications demandent la même quantité de temps, ce qui n'est pas toujours le cas. Tout ce qu'on peut valablement donner, c'est un intervalle de performances exprimées en LIPS.

Jusqu'ici, nous n'avons évoqué que l'aspect *vitesse d'exécution* des programmes. Un autre aspect important à envisager est celui de l'espace mémoire. Il faut s'intéresser à la fois à la taille du code généré et à la mémoire nécessaire pour l'exécution du programme.

Dans le cadre des systèmes-experts, l'espace mémoire utilisé est souvent une donnée cruciale car les moteurs d'inférences en sont de grands consommateurs.

Nous avons écrit, utilisé ou aménagé des programmes permettant de comparer des systèmes-experts vides possédant des correspondances bien définies concernant leurs spécificités.

Les résultats obtenus par l'application de ces programmes ne fournissent cependant que des indications sur le comportement d'un produit; ils ne permettent en aucun cas d'affirmer que, *globalement*, tel produit est meilleur que tel autre.

Ces programmes ne testent qu'une partie de l'aspect **dynamique** d'un produit. Par aspect dynamique, nous entendons tout ce qui est mesurable, comme la vitesse d'exécution, la demande en taille mémoire, ...

A cet aspect, il faut ajouter l'aspect **statique**. Chaque langage, chaque implémentation possède ses propres *fonctionnalités*, son petit *plus* par rapport à la concurrence.

Comparer des implémentations sur base de leurs fonctionnalités nous paraît primordial. Si un produit ne supporte pas les caractéristiques exigées pour sa mise en application, les questions d'efficacité n'ont plus beaucoup d'intérêt.

Pour subjectif que soit le choix d'un langage, il nous paraît important de tenir compte de ces deux aspects. C'est pourquoi nous accompagnerons de quelques considérations **statiques** la présentation de chaque produit .

Dans cette étude, nous avons analysé deux familles de systèmes-experts vides : KOOL et PROLOG.

Nous avons eu l'occasion d'étudier KOOL lors d'un stage de quatre mois effectué chez Bull, à Paris, fin 1985.

En ce qui concerne Prolog, l'intérêt de plus en plus grand qu'il suscite nous a donné le désir de l'analyser. Notre choix s'est porté sur un IBM PC parce qu'il existe

aujourd'hui une demi-douzaine d'implémentations disponibles sous MS/DOS (nous n'avons malheureusement pu disposer que de quelques-unes d'entre elles) et que nous possédions un micro-ordinateur IBM.

Notre but est de fournir les éléments d'une démarche pour permettre à un développeur de choisir une implémentation donnée pour une application donnée suivant des critères de performance et de fonctionnalité.

CHAPITRE II : Le système-expert vide KOOL

II.1. Introduction au système-expert KOOL

Le système-expert **KOOL** (*Knowledge representation Object Oriented Language*), développé au Centre de Recherche BULL et écrit en LISP, permet de déclarer des objets munis d'attributs, des règles d'inférences portant sur ces attributs, et de spécifier des actions (création de nouveaux objets, envoi de messages aux objets...).

Pur produit de recherche à l'origine, la validation de KOOL en tant qu'outil d'usage "industriel" restait à faire. En particulier, les aspects limites et performances étaient critiques:

- nombre maximum de règles, de classes d'objets et d'instances définissables,
- mémoire et temps CPU consommés,
- variation des performances selon les *directions propres* du produit: nombre de règles/d'objets/profondeur de récursivité, ...

II.2. Analyse de l'architecture de KOOL

II.2.1. Introduction

Le système-expert vide KOOL a été développé par le Centre de Recherche Bull sur Multics[1] en LE_LISP. Il inclut un langage de programmation destiné à l'écriture de systèmes experts. KOOL réalise une "*tentative de synthèse de trois paradigmes classiques (...): les objets, les frames et les règles de production*" [Launet 85].

A ce titre, KOOL appartient à la famille des Langages Orientés Objets (LOO). La programmation orientée objet poursuit les mêmes buts que la programmation structurée, mais

[1] **Multics**: système d'exploitation développé en PL/I au M.I.T. dès 1964. En 1965, les Bell Telephone Laboratories et le département informatique de la General Electric Company se sont joints au projet. Ce système a fortement inspiré Unix.

s'adapte à des problèmes de complexité plus grande. Ces langages permettent une programmation plus fonctionnelle qu'algorithmique.

KOOL, comme ces *Langages Orientés Objet*, possède un certain nombre de concepts:

a. L'**objet** : permet de représenter en une seule entité les données et les procédures d'exploitation. Il va servir à donner une description structurale et une description de la dynamique de l'univers.

b. La **classe** : permet de regrouper des objets de descriptions semblables, chacun de ces objets étant une instance de la classe. Ces objets peuvent être instantiés dynamiquement.

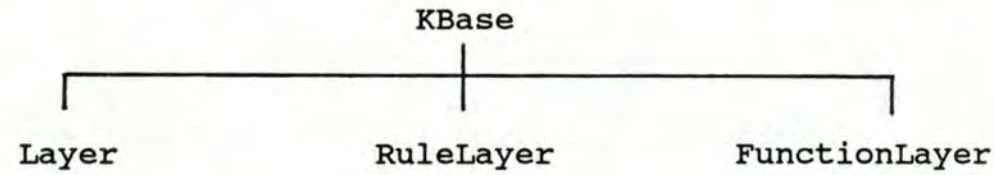
c. L'**héritage** : une classe peut être créée à partir d'une classe existante. Il existe alors une hiérarchie entre les classes : les sous-classes héritent des propriétés de la sur-classe, qui s'ajoutent à leurs propres propriétés.

d. Les **messages** et les **méthodes** : les objets communiquent par l'envoi de messages. Un message est constitué d'un nom désignant le type de manipulation désiré, de l'objet destinataire et de divers paramètres. En envoyant un message, le programmeur décrit ce qu'il désire, mais non comment l'obtenir. A la réception d'un message le destinataire détermine la procédure - le programme - que nous appellerons désormais *méthode*, à activer. Chaque méthode peut être définie localement ou être héritée via la hiérarchie des classes.

e. Les **attributs** : ce sont des *variables* qui caractérisent chaque classe.

II.2.2. Représentation de la connaissance

La structure de la description de la connaissance dans KOOL peut être schématisée comme suit :



La classe **KBase** (pour *Knowledge Base*) contient la liste de toutes les *Layer* d'une application. Les messages auxquels les instances de cette classe savent répondre permettent le chargement, la sauvegarde et la sélection d'une KBase courante. Plusieurs KBases peuvent être regroupées dans une *KBBase*.

La classe **Layer** contiendra toutes les classes et objets de l'application. Une *Layer* peut être vue comme une couche qui regroupe des sous-éléments. Tout comme pour la KBase, il est possible de charger, sauvegarder et sélectionner une occurrence de la classe *Layer*.

La classe **RuleLayer** va servir à répertorier les définitions des règles de production. Il est possible de charger et de sauver une RuleLayer donnée et les fonctions Lisp associées aux règles. De plus, le système peut charger le code source des règles dans l'environnement .

La classe **FunctionLayer** va rassembler les définitions de fonctions.

Pour écrire une application en KOOL, il faudra donc se choisir une KBase, des classes et donc une Layer, des règles et une RuleLayer, des fonctions et une FunctionLayer.

Dans une **classe**, on trouvera la description du *Meta*, des *Superclasses*, des *Cvars*, des *Ivars* et des *Methods*.

Le descripteur **Meta** est associé à tout objet connu. Il relie cet objet à sa classe.

Le descripteur **Superclasses** permet de définir le lien de type hiérarchique qui traduit l'inclusion de l'ensemble défini par cette classe dans celui défini par sa superclasse.

Par exemple, si on souhaite exprimer que *les humains sont des mortels*, on écrira :

```
Humain
     meta : Class
     supers : (Mortel)    | les humains sont des mortels.
Mortel
     meta : Class
     ...
```

Les **Cvars** permettent de définir des attributs qui ont la même valeur pour tous les objets appartenant à la classe.

Par exemple, quand on dit que *les humains sont mortels*, le trait *mortel*, associé à la classe des humains a toujours la valeur de vérité vraie. On traduira cela en KOOL par :

```
Humain
     meta : Class
     cvars : (mortel)     | mortel est une "variable" de
                          | classe.
     mortel : t           | cette variable a la valeur
                          | t (vraie).
```

Les **Ivars** sont des variables d'instances. Ce sont des attributs dont les valeurs peuvent être différentes suivant l'objet de la classe. On associe à chaque attribut d'instance un domaine qui exprime une restriction sur les valeurs possibles de cet attribut.

Par exemple, pour traduire qu'*un homme se marie avec une femme*, on écrira :

```
Homme
     meta : Class
     ivars : (marié_avec)| un homme peut se marier.
     marié_avec : nil    | on ne connaît pas sa femme.
       type : (any Femme)|
```

Les **Methods** indiquent les fonctions qui vont répondre aux messages. Une méthode est une fonction Lisp qui est associée à un sélecteur.

Pour dire qu'*un robot peut prendre ou poser un cube*, on écrira :

```
Robot
     meta : Class
     methods :
       prendre : prendrecube | prendrecube et posercube
       poser : posercube     | sont deux fonctions Lisp.
(de prendrecube ...)
(de posercube ...)
```

Il existe en KOOL un certain nombre de descripteurs prédéfinis :

- le **type** permet de définir une restriction sur l'ensemble des valeurs autorisées,
- la **relation** indique qu'à l'attribut correspond non pas une valeur mais une liste de valeurs,
- le **meaning** (signification) est une chaîne de caractères qui sera imprimée quand on demandera à l'utilisateur d'entrer la valeur de l'attribut,
- le **askable** (demandable) permet de savoir si, lorsqu'on cherche à déterminer la valeur de l'attribut, on peut demander à l'utilisateur de donner la valeur de cet attribut,
- la **question** est une chaîne de caractères qui sera imprimée lorsqu'on demandera à l'utilisateur la valeur de l'attribut,
- le **default** (défaut) permet d'assigner une valeur par défaut à l'attribut.
- le **init** est une expression qui sera évaluée lors de la création d'un représentant de la classe et qui est associée à l'attribut.
- le **to-fill** permet d'associer à un attribut une expression qui sera évaluée si, quand on recherchera la valeur de l'attribut, on ne la trouve pas,
- le **when-filled** permet de demander que, lorsqu'on associe une valeur à un attribut, toutes les expressions déclarées ici soient évaluées,
- le **when-removed** fonctionne comme le *when-filled* si ce n'est que l'évaluation s'effectuera lors de la suppression de la valeur de l'attribut.

II.2.3. Le moteur d'inférences

En KOOL, la description de **la dynamique de l'univers** se fait par des *règles de production*. Elles se présentent classiquement ici sous forme de clauses "*IF ... THEN ...*". Les prémisses et les conclusions peuvent porter sur les attributs et faire appel à la messagerie. Les règles sont aussi des objets (instances de la classe Rule), ce qui permet leur intégration naturelle dans le reste du langage.

On dispose, pour les règles, des fonctionnalités suivantes :

- **chargement** en mémoire d'un ensemble de règles, c'est-à-dire du texte source de la règle et des fonctions Lisp associées,
- **sauvegarde** sur disque d'un ensemble de règles,
- **installation** des règles dans l'environnement (en fait, l'association de la règle à une RuleLayer),
- **affichage** du code source d'une règle (cette fonction n'était pas implémentée dans la version testée),
- **édition** d'une règle à l'aide d'un éditeur du système hôte (cette fonction n'était pas implémentée dans la version testée),
- **traçage** d'une ou de toutes les règles avec affichage des tentatives de déclenchement et des déclenchements effectifs (cette fonction n'était pas implémentée dans la version testée).

Le fonctionnement des règles peut se faire en chaînage-avant ou en chaînage-arrière.

Le **chaînage-avant** correspond à un fonctionnement de type *propagation*. Lors d'une assertion (création d'un objet ou association d'une valeur à un attribut), le système dérive immédiatement les conclusions définies par les règles qui mentionnent cet attribut. Si le système trouve un ensemble consistant d'instantiations pour les variables figurant dans la règle, c'est-à-dire un ensemble de substitutions variable/instance tel que toutes les prémisses soient satisfaites, la règle est déclenchée. Ce déclenchement a deux conséquences :

1) évaluation de gauche à droite des clauses figurant dans la partie conclusion de la règle. Pendant cette évaluation, le système est gelé, c'est-à-dire que les règles susceptibles de réagir à l'une des assertions

figurant dans les conclusions de la règle déclenchée ne seront prises en compte que lorsque toutes les conclusions auront été effectuées;

2) passage de l'univers dans l'état suivant.

Le mode de propagation choisi est le mode *en profondeur d'abord*. Dans ce mode de propagation, si on décrit l'ensemble des règles applicables comme un arbre, cet arbre sera parcouru dans le sens *branche gauche -> noeud -> branche droite*. Ce mode permet une réaction immédiate du système sur l'assertion la plus récente. Dès qu'une modification intervient, l'ensemble des règles qui la prennent en compte est inspecté et réagit à ces modifications. On dispose ainsi d'un système extrêmement réactif à tout environnement.

Le **chaînage-arrière** peut être interprété comme la sélection d'un sous-ensemble des règles du système qui permet d'aboutir à la déduction de la valeur d'un attribut. Un attribut est dit *déductible* s'il est inconnu ou si la règle concluant sur la valeur de cet attribut a été définie utilisable en mode *to_fill*.

Le mode standard est le parcours exhaustif des règles concluant sur l'attribut considéré. La stratégie de sélection des règles est également en profondeur d'abord. Si dans l'évaluation d'une prémisse la valeur d'un attribut est inconnue, le sous-but "*quelle est la valeur de cet attribut*" est lancé. Le système examine alors les règles concluant sur cet attribut, ceci pouvant récursivement lancer de nouveaux sous-buts pour déduire la valeur d'autres attributs dont la connaissance est nécessaire à la détermination du précédent.

Il est à remarquer que le parcours exhaustif de toutes les règles n'est pas systématiquement nécessaire. En effet, dans le cas de l'attribution d'une valeur à un attribut de type propriété, on peut se contenter de ne déclencher qu'une seule règle. Dans le cas d'un attribut de type *relation*, il est nécessaire de parcourir toutes les règles car on souhaite ici connaître l'ensemble des valeurs associées à l'attribut.

L'utilisateur est libre de décider du mode de résolution. Il lui suffit de décider que tel attribut sera utilisé en mode chaînage-avant et tel autre en mode chaînage-arrière.

En résumé, KOOL permet de décrire des objets et de définir leur comportement. Un objet est défini statiquement par un ensemble d'attributs qui le caractérisent. La dynamique de l'univers modélisé est représentée par des règles de production et des comportements associés à des classes d'objets.

II.3. Relevé des variables d'états

II.3.1 Introduction

L'utilisation de KOOL et l'analyse du code LE_LISP ont permis de dégager un certain nombre de variables susceptibles d'avoir une influence sur le temps CPU et la quantité de mémoire utilisée. Ces variables (dites variables d'état) sont, par ordre alphabétique :

- l'attribut
- la classe,
- la conclusion,
- le descripteur,
- la FunctionLayer,
- la fonction,
- l'instance,
- la KBase,
- la Layer,
- la prémisse,
- la RuleLayer.
- la règle,
- le sélecteur,

II.3.2 Liste des principales variables d'état triées en ordre décroissant sur le coût CPU

Toutes autres choses restant égales, une augmentation substantielle d'une des variables suivantes paraît de moins en moins coûteuse en ce qui concerne le temps CPU consommé:

- l'instance: le nombre d'unifications va augmenter; on travaille sans cesse sur les instances.
- la prémisse: à chaque activation du moteur, on va essayer de vérifier les prémisses de toutes les règles. Le nombre de tests va donc augmenter.

- la conclusion: plus les conclusions seront nombreuses, plus le nombre d'actions (création, modification, destruction, ...) va augmenter.
- le descripteur: l'utilisation de descripteurs s'avère coûteuse car un descripteur nécessite des contrôles de type, des demandes à l'utilisateur, la maintenance des relations, ...
- la classe: elle va contenir des informations nombreuses et mouvantes qu'il faudra mettre à jour.
- la fonction: le temps CPU consommé est en rapport avec la complexité de la fonction.

II.3.3. Liste des principales variables d'état triées en ordre décroissant sur le coût mémoire

Toutes autres choses restant égales, une augmentation substantielle d'une des variables suivantes paraît de moins en moins coûteuse en ce qui concerne la mémoire centrale consommée:

- l'instance,
- la classe,
- la règle,
- la prémisse,
- la conclusion.

II.4. Un langage de génération automatique d'environnement KOOL

On souhaite disposer d'un langage qui permette la création automatique d'un environnement, tant en ce qui concerne les objets que les règles. Ce langage doit permettre de faire varier le nombre et le type des différents objets manipulés dans un objectif restreint.

II.4.1. Les fonctions

L'utilisateur peut créer un nombre quelconque de fonctions (jusqu'à saturation de la mémoire). Toutes les fonctions sont identiques, seul leur nom varie. Les fonctions créées acceptent un paramètre numérique en entrée. Elles décrémentent ce paramètre par pas de un jusque zéro. Le type de fonction n'a guère d'importance ici, d'autant plus qu'en LE_LISP, les fonctions récursives n'ont aucune incidence sur la hauteur de la pile. En effet, certains interpréteurs Lisp

gèrent de manière itérative certains types de récursions [Greussay 77]. Il ne s'agit pas d'une transformation syntaxique ou sémantique, mais bien d'une gestion spéciale des ressources nécessaires à l'évaluation. Les récursions terminales sont traitées comme des itérations et leur exécution s'effectue en espace constant.

II.4.2. Les classes

Pour chaque classe, le langage permet à l'utilisateur de spécifier un certain nombre de paramètres:

- le nombre de classes,
- le nombre d'instances,
- le nombre de variables de classes,
- le nombre de variables d'instances.

II.4.3. Les règles

Pour chaque règle, on pourra spécifier les paramètres suivants:

- le nombre de prémisses portant sur des variables de classes,
- le nombre de prémisses portant sur des variables d'instances,
- le nombre de prémisses faisant appel à la hiérarchie,
- le nombre de prémisses appelant une fonction,
- le nombre de conclusions portant sur des variables d'instances,
- le nombre de conclusions créant une instance,
- le nombre de conclusions appelant la messagerie,
- le nombre de conclusions appelant une fonction,
- le code d'application de la règle (en chaînage-avant (**w**) ou en chaînage-arrière et avant (**b**)).

Les variables sur lesquelles portent les règles sont choisies de manière déterministe afin d'assurer une couverture maximale de toutes les classes.

Par contre, les fonctions appelées sont choisies au hasard.

II.5. Analyse de l'instrumentation

Le système hôte LE_LISP met à la disposition de l'utilisateur des fonctions de mesure. Nous avons ici utilisé les fonctions RUNTIME et GC.

La fonction RUNTIME calcule le temps CPU utilisé depuis l'appel du système LE_LISP.

La fonction GC appelle explicitement le *garbage-collector* et retourne une liste contenant les informations relatives à la dernière récupération de mémoire.

II.6. Les jeux de tests

L'idéal eut été de tester le système avec "toutes" les combinaisons possibles des variables d'état et avec des valeurs proches de celles rencontrées dans un problème réel. Malheureusement, des contraintes de temps ont empêché ces mesures "exhaustives". De plus, les premiers tests ont révélé des problèmes de limite de taille de la pile LE_LISP. Le plan de test a alors été réactualisé. Les mesures ont porté sur les combinaisons des valeurs suivantes (voir tableau 6.1).

infér	inst	class	règle	cvar	p_cvar	p_ivar	c_ivar	ivar	hiérar	msg	code
5	5	1	1	0	0	1	1	5	0	0	w
10	50	5	2	5	1	2	2	10	5	1	b
20	250	10	5	10	2	5	5	20	10	2	
				20	4				20	5	

tableau 6.1. jeux de tests

Explication des abréviations :
- *infér* : nombre d'inférences,
- *inst* : nombre d'instances de chaque classe,
- *class* : nombre de classes,
- *règle* : nombre de règles,
- *cvar* : nombre de variables de classes par classe,
- *p_cvar* : nombre de prémisses portant sur les variables de classes,
- *p_ivar* : nombre de prémisses portant sur les variables d'instances,
- *c_ivar* : nombre de conclusions portant sur les variables d'instances,
- *ivar* : nombre de variables d'instances par classe,
- *hiérar* : nombre d'appels à la hiérarchie,
- *msg* : nombre d'appels à la messagerie,
- *code* : application des règles en chaînage-avant (w) ou arrière et avant (b).

II.7 Les résultats des mesures

Signification des abréviations :

- N : numéro du test,
- Cl : nombre de classes,
- Ins : nombre d'instances,
- Cv : nombre de variables de classes,
- Iv : nombre de variables d'instances,
- Ru : nombre de règles,
- PC : nombre de prémisses sur variable de classe,
- PI : nombre de prémisses sur variable d'instance,
- PH : nombre de prémisses appelant la hiérarchie,
- PF : nombre de prémisses appelant une fonction,
- CI : nombre de conclusions sur variable d'instance,
- Ci : nombre de conclusions créant une instance de classe,
- CM : nombre de conclusions appelant la messagerie,
- CF : nombre de conclusions appelant une fonction,
- C : code d'application de la règle,
- In : nombre d'inférences,
- CPU : temps CPU en secondes,
- con : nombre de *CONS** consommés ,
- sy : nombre de *SYMBOL** consommés,
- st : nombre de *STRING** consommés,
- fl : nombre de *FLOAT** consommés.

Les termes marqués d'une * seront expliqués lors de l'interprétation des résultats.

*N	*Cl	Ins	Cv	Iv	Ru	PC	PI	PH	PF	CI	Ci	CM	CF	C	In	CPU	con	sy	st	fl*
* 1*	1	**5**	0	5	1	0	1	0	0	1	0	0	0	w	*5*	0,296	137	2	20	2*
* 2*	1	**5**	0	5	1	0	1	0	0	1	0	0	0	w	*10*	0,440	172	2	20	2*
* 3*	1	**5**	0	5	1	0	1	0	0	1	0	0	0	w	*20*	0,860	242	2	20	2*
* 4*	1	**50**	0	5	1	0	1	0	0	1	0	0	0	w	*5*	0,300	137	2	20	2*
* 5*	1	**50**	0	5	1	0	1	0	0	1	0	0	0	w	*10*	0,450	172	2	20	2*
* 6*	1	**50**	0	5	1	0	1	0	0	1	0	0	0	w	*20*	0,780	242	2	20	2*
* 7*	1	**250**	0	5	1	0	1	0	0	1	0	0	0	w	*5*	0,300	137	2	20	2*
* 8*	1	**250**	0	5	1	0	1	0	0	1	0	0	0	w	*10*	0,498	172	2	20	2*
* 9*	1	**250**	0	5	1	0	1	0	0	1	0	0	0	w	*20*	0,780	242	2	20	2*
10	1	5	0	**10**	1	0	1	0	0	1	0	0	0	w	*5*	0,510	176	2	20	2*
11	1	5	0	**10**	1	0	1	0	0	1	0	0	0	w	*10*	0,536	218	2	20	2*
12	1	5	0	**10**	1	0	1	0	0	1	0	0	0	w	*20*	1,287	288	2	20	2*
13	1	5	0	**20**	1	0	1	0	0	1	0	0	0	w	*5*	0,901	182	2	20	2*
14	1	5	0	**20**	1	0	1	0	0	1	0	0	0	w	*10*	0,814	217	2	20	2*
15	1	5	0	**20**	1	0	1	0	0	1	0	0	0	w	*20*	1,027	287	2	20	2*
16	**5**	5	0	5	1	0	1	0	0	1	0	0	0	w	*5*	0,302	161	2	20	2*
17	**5**	5	0	5	1	0	1	0	0	1	0	0	0	w	*10*	0,467	196	2	20	2*
18	**5**	5	0	5	1	0	1	0	0	1	0	0	0	w	*20*	0,764	266	2	20	2*
19	**10**	5	0	5	1	0	1	0	0	1	0	0	0	w	*5*	0,333	137	2	19	2*
20	**10**	5	0	5	1	0	1	0	0	1	0	0	0	w	*10*	0,479	172	2	19	2*
21	**10**	5	0	5	1	0	1	0	0	1	0	0	0	w	*20*	1,177	242	2	19	2*
22	1	5	**5**	5	1	1	1	0	0	1	0	0	0	w	*5*	0,411	137	2	20	2*
23	1	5	**5**	5	1	1	1	0	0	1	0	0	0	w	*10*	0,523	172	2	20	2*
24	1	5	**5**	5	1	1	1	0	0	1	0	0	0	w	*20*	0,961	266	2	20	2*
25	1	5	**10**	5	1	1	1	0	0	1	0	0	0	w	*5*	0,420	137	2	20	2*
26	1	5	**10**	5	1	1	1	0	0	1	0	0	0	w	*10*	0,516	172	2	20	2*
27	1	5	**10**	5	1	1	1	0	0	1	0	0	0	w	*20*	0,980	211	2	20	2*
28	1	5	**20**	5	1	1	1	0	0	1	0	0	0	w	*5*	0,351	161	2	20	2*
29	1	5	**20**	5	1	1	1	0	0	1	0	0	0	w	*10*	0,617	196	2	20	2*
30	1	5	**20**	5	1	1	1	0	0	1	0	0	0	w	*20*	1,220	266	2	20	2*

N	Cl	Ins	Cv	Iv	Ru	PC	PI	PH	PF	CI	Ci	CM	CF	C	In	CPU	con	sy	st	fl
31	1	5	5	5	1	**2**	1	0	0	1	0	0	0	w	*5*	0,415	137	2	20	2*
32	1	5	5	5	1	**2**	1	0	0	1	0	0	0	w	*10*	0,594	196	2	20	2*
33	1	5	5	5	1	**2**	1	0	0	1	0	0	0	w	*20*	1,178	266	2	20	2*
34	1	5	5	5	1	**4**	1	0	0	1	0	0	0	w	*5*	0,443	161	2	20	2*
35	1	5	5	5	1	**4**	1	0	0	1	0	0	0	w	*10*	0,738	196	2	20	2*
36	1	5	5	5	1	**4**	1	0	0	1	0	0	0	w	*20*	1,431	266	2	20	2*
37	1	5	0	5	1	0	1	**5**	0	1	0	0	0	w	*5*	0,383	167	2	20	2*
38	1	5	0	5	1	0	1	**5**	0	1	0	0	0	w	*10*	0,612	202	2	20	2*
39	1	5	0	5	1	0	1	**5**	0	1	0	0	0	w	*20*	1,189	272	2	20	2*
40	1	5	0	5	1	0	1	**10**	0	1	0	0	0	w	*5*	0,465	143	2	20	2*
41	1	5	0	5	1	0	1	**10**	0	1	0	0	0	w	*10*	0,884	178	2	20	2*
42	1	5	0	5	1	0	1	**10**	0	1	0	0	0	w	*20*	-	-	-	-	-*
43	1	5	0	5	1	0	1	**20**	0	1	0	0	0	w	*5*	0,694	143	2	20	2*
44	1	5	0	5	1	0	1	**20**	0	1	0	0	0	w	*10*	-	-	-	-	-*
45	1	5	0	5	1	0	1	**20**	0	1	0	0	0	w	*20*	-	-	-	-	-*
46	5	5	0	5	**2**	0	1	0	0	1	0	0	0	w	*5*	0,296	161	2	21	2*
47	5	5	0	5	**2**	0	1	0	0	1	0	0	0	w	*10*	0,449	196	2	21	2*
48	5	5	0	5	**2**	0	1	0	0	1	0	0	0	w	*20*	0,767	266	2	21	2*
49	5	5	0	5	**5**	0	1	0	0	1	0	0	0	w	*5*	0,713	221	6	15	2*
50	5	5	0	5	**5**	0	1	0	0	1	0	0	0	w	*10*	1,128	256	6	15	2*
51	5	5	0	5	**5**	0	1	0	0	1	0	0	0	w	*20*	1,223	326	6	15	2*
52	5	5	0	5	1	0	**2**	0	0	1	0	0	0	w	*5*	0,416	161	2	20	2*
53	5	5	0	5	1	0	**2**	0	0	1	0	0	0	w	*10*	0,508	196	2	20	2*
54	5	5	0	5	1	0	**2**	0	0	1	0	0	0	w	*20*	0,832	266	2	20	2*
55	5	5	0	5	1	0	**5**	0	0	1	0	0	0	w	*5*	0,440	161	2	20	2*
56	5	5	0	5	1	0	**5**	0	0	1	0	0	0	w	*10*	0,741	196	2	20	2*
57	5	5	0	5	1	0	**5**	0	0	1	0	0	0	w	*20*	0,921	266	2	20	2*
58	5	5	0	5	1	0	1	0	0	**2**	0	0	0	w	*5*	0,401	191	2	20	2*
59	5	5	0	5	1	0	1	0	0	**2**	0	0	0	w	*10*	0,682	256	2	20	2*
60	5	5	0	5	1	0	1	0	0	**2**	0	0	0	w	*20*	1,393	386	2	20	2*

N	Cl	Ins	Cv	Iv	Ru	PC	PI	PH	PF	CI	Ci	CM	CF	C	In	CPU	con	sy	st	fl
61	5	5	0	5	1	0	1	0	0	**5**	0	0	0	w	*5*	0,890	311	2	20	2*
62	5	5	0	5	1	0	1	0	0	**5**	0	0	0	w	*10*	1,442	436	2	20	2*
63	5	5	0	5	1	0	1	0	0	**5**	0	0	0	w	*20*	2,760	746	2	20	2*
64	1	5	0	5	1	0	1	0	0	1	0	**1**	0	w	*5*	0,352	137	2	20	2*
65	1	5	0	5	1	0	1	0	0	1	0	**1**	0	w	*10*	0,544	172	2	20	2*
66	1	5	0	5	1	0	1	0	0	1	0	**1**	0	w	*20*	1,103	242	2	20	2*
67	1	5	0	5	1	0	1	0	0	1	0	**2**	0	w	*5*	0,477	137	2	20	2*
68	1	5	0	5	1	0	1	0	0	1	0	**2**	0	w	*10*	0,699	172	2	20	2*
69	1	5	0	5	1	0	1	0	0	1	0	**2**	0	w	*20*	1,335	242	2	20	2*
70	1	5	0	5	1	0	1	0	0	1	0	**5**	0	w	*5*	0,580	137	2	20	2*
71	1	5	0	5	1	0	1	0	0	1	0	**5**	0	w	*10*	0,950	172	2	20	2*
72	1	5	0	5	1	0	1	0	0	1	0	**5**	0	w	*20*	1,775	266	2	20	2*
73	1	5	0	5	1	0	1	0	0	1	0	0	0	**b**	*5*	0,302	137	2	20	2*
74	1	5	0	5	1	0	1	0	0	1	0	0	0	**b**	*10*	0,591	172	2	20	2*
75	1	5	0	5	1	0	1	0	0	1	0	0	0	**b**	*20*	0,919	242	2	20	2*

II.8 Interprétation des résultats

II.8.1. Introduction

Avertissement: *des mesures répétées sur les mêmes tests et avec les mêmes paramètres ont montré que les temps CPU donnés par le système LE_LISP/MULTICS sont corrects avec un écart de 5 à 10 pour cent.*

Dans l'ensemble des tests, on remarque un coût "fixe" et un coût "variable". Une partie des tests donne les résultats attendus (en général une augmentation du temps CPU avec l'augmentation de la valeur d'un paramètre) tandis que d'autres donnent des résultats non prévus.

II.8.2. Interprétations des résultats du point de vue du temps CPU

II.8.2.1. Le nombre d'inférences

L'augmentation linéaire du nombre d'inférences produit une augmentation linéaire du temps de réponse. Ceci semble normal car le système exécute plusieurs fois un programme identique.

II.8.2.2. Le nombre d'instances

La croissance du nombre d'instances n'a quasi pas d'influence au niveau du temps CPU. Ce résultat est heureux, mais on aurait pu s'attendre à une augmentation du temps CPU consommé car le nombre moyen de tentatives d'unifications augmente.

II.8.2.3. Le nombre de variables d'instances

A une variation du nombre de variables d'instances correspond une variation linéaire du temps de réponse associé . La variation devient cependant plus cahotique pour 20 inférences.

Il semble normal que le temps CPU augmente avec l'augmentation du nombre de variables d'instances puisque le système doit parcourir un espace mémoire plus important.

II.8.2.4. Le nombre de classes

Le temps CPU augmente très légèrement et de manière linéaire avec l'augmentation du nombre de classes. En effet, le moteur doit parcourir un espace plus important pour trouver la classe qu'il recherche.

II.8.2.5. Le nombre de variables de classes

L'introduction de variables de classes dans les règles ralentit l'exécution d'environ 30 pour cent. D'autre part, une augmentation du nombre de c_vars provoque une augmentation linéaire, dans un rapport deux, du temps CPU. Cette augmentation est normale car les règles deviennent plus complexes. De plus, l'augmentation n'est pas exagérée car le système doit parfois dériver la valeur d'une variable de classe en parcourant la hiérarchie.

II.8.2.6. Le nombre d'appels à la hiérarchie

Les appels à la hiérarchie provoquent un accroissement d'environ 30 pour cent du temps de réponse du système. La variation du nombre d'appels à la hiérarchie provoque une variation linéaire, d'un rapport trois, du temps CPU.

II.8.2.7. Le nombre d'attributs des classes

Le fait d'augmenter le nombre d'attributs des classes provoque un comportement bizarre du système. Suivant le nombre d'inférences, tantôt l'un tantôt l'autre type de variable d'instance donne des résultats meilleurs. Nous ne voyons pas d'explication à ce comportement.

II.8.2.8. Le nombre de règles

L'augmentation du nombre de règles fait apparaître une croissance exponentielle du temps de réponse. En effet, en augmentant le nombre de règles, on oblige le système à parcourir un plus grand espace mémoire avant de trouver la règle qui l'intéresse.

Cette tendance s'atténue pourtant avec l'augmentation du nombre d'inférences. Nous ne voyons pas d'explication à ce phénomène.

II.8.2.9. Le nombre de prémisses

La variation du nombre de prémisses portant sur les variables d'instances produit des résultats anarchiques auxquels nous ne voyons pas d'explication.

II.8.2.10. Le nombre de conclusions

Une augmentation du nombre de conclusions sur les variables d'instances provoque une augmentation très sensible du temps de réponse CPU. Ce comportement semble normal puisque, lorsqu'une règle est déclenchée, il faut effectuer un plus grand nombre d'actions.

II.8.2.11. Le nombre d'appels à la messagerie

L'introduction d'appels à la messagerie dans les conclusions entraîne un surcroît de 15 pour cent du temps d'exécution.

II.8.2.12. Le mode d'application des règles

L'application des règles en chaînage-avant et arrière s'avère un peu plus coûteuse qu'une exécution en chaînage-avant seulement. L'écart semble cependant diminuer si le nombre d'inférences augmente.

II.8.3. Interprétation des résultats du point de vue espace mémoire

II.8.3.1. La pile LE_LISP

La hauteur de la pile LE_LISP/MULTICS est limitée à 535 éléments. La hauteur de départ, avant toute application de règle, est de 38 éléménts. Une règle sans prémisse et sans conclusion coûte à chaque appel 6 éléments de pile. L'introduction d'une prémisse ou d'une conclusion exige 3 éléments de pile. Connaissant le nombre et le type de règles, on peut dès lors calculer le plus grand problème soluble par KOOL dans cet environnement :

```
max = 535 - 38 - {(#règles * 3 * (#prémisses + #conclusions)
* 3)}
```

II.8.3.2. Le nombre de CONS

Le **nombre de CONS** est le nombre de cellules de liste restant dans la zone des listes. On dispose de plus de 21.800 cellules dans le système utilisé.

Une instance d'une classe contenant 5 variables d'instance possédant un attribut coûte en moyenne 27 cellules. L'ajout d'une variable d'instance représente 8 cellules supplémentaires par instance de la classe. L'ajout d'une variable de classe, si elle représente le même coût de 8 cellules, ne demande ces cellules qu'une fois par classe. Une instance supplémentaire d'une classe possédant 5 variables d'instance nécessite 40 cellules. L'ajout d'une classe du type ci-dessus représente 296 cellules.

Une règle d'une prémisse et d'une conclusion représente 94 cellules de listes. L'ajout d'une prémisse ou d'une conclusion nécessite 19 cellules supplémentaires.

II.8.3.3. Le nombre de SYMBOLS

Le **nombre de SYMBOLS** est le nombre de symboles restant disponibles dans la zone des symboles. Au départ, on dispose de place pour 3.600 symboles.

Il faut compter un symbole par variable, par classe et par instance. De même, on comptera un symbole par prémisse, par conclusion et par règle.

II.8.3.4. Le nombre de STRINGS

Le **nombre de STRINGS** est le nombre de chaînes restant dans la zone chaîne. On dispose de 628 chaînes au départ. La consommation de chaînes se calcule de la même façon que la consommation de symboles.

II.9 Conclusions

Pour autant que l'on puisse en juger, KOOL paraît performant. Il est puissant, souple et riche. Ses qualités font de lui un outil polyvalent, susceptible de s'adapter à tout type de problème. De plus, les programmes rédigés en KOOL sont faciles à lire et reflètent clairement le monde modélisé. Sa plus grosse limitation semble provenir de sa voracité en éléments de la pile LISP. Ceci pourrait s'expliquer par le fait que le "conflict-set" est mémorisé dans la pile.

La représentation physique des objets n'est pas optimale, mais le concepteur estime que des modifications actuellement à l'étude devraient permettre de remédier à cet état de choses.

Reste cependant le comportement inattendu du système face à certaines situations. On pensera plus particulièrement aux variations du nombre d'attributs des classes ou encore aux variations du nombre de prémisses portant sur les variables d'instances.

CHAPITRE III : Le langage PROLOG

III.1. Introduction au langage PROLOG

III.1.1. Historique

En octobre 1981, le fameux Miti, ministère Japonais du commerce international et de l'industrie, invita quelques-uns des principaux responsables occidentaux de la recherche et du développement en informatique. Ce que l'on voulait présenter à cet auditoire prestigieux mais quelque peu ébahi était un ambitieux plan de développement de systèmes de « cinquième génération », impliquant une remise en cause des notions actuelles sur le matériel comme sur le logiciel. Pour ce qui est du logiciel, justement, le projet présenté indiquait que tout reposerait sur un certain langage, Prolog. On peut penser que plus d'une parmi les personnalités invitées s'empressa, à son retour du colloque, de convoquer ses subordonnés et les enjoignit de découvrir ce que recouvrait ce nom magique.

C'est ainsi qu'un langage développé par quelques chercheurs en intelligence artificielle, et connu jusque-là d'un seul public de spécialistes, se trouva brusquement projeté sous les feux de l'actualité. [Colmerauer et al. 83]

Le langage PROLOG a été mis au point aux alentours des années 1970 par **Alain Colmerauer** et ses collègues de la faculté des sciences de Luminy à Marseille dans le cadre d'une tentative de compréhension automatique du langage naturel. C'était une première tentative de conception d'un langage qui offre au programmeur la possibilité d'exprimer des tâches en logique plutôt qu'en termes de la programmation traditionnelle, où l'on indique à la machine *que* faire et *quand*. Cet objectif explique le nom de ce langage de programmation : **PROLOG** veut dire **PRO**grammation **LOG**ique. Prolog a essaimé dans le monde et il n'existe pas de standard officiel Prolog. On distingue en fait plusieurs

implémentations qui sont devenues des standards de facto. Ces implémentations diffèrent essentiellement par leur terminologie et leur syntaxe. On retiendra spécialement le Prolog Marseille, le Prolog Edinburgh (le premier à incorporer un compilateur), le Prolog Waterloo (Canada) et la version hongroise M-Prolog.

Prolog abandonne le principe de la programmation impérative (où l'informaticien indique à l'ordinateur l'algorithme à appliquer pour calculer le résultat qu'il veut obtenir) au profit de la programmation déclarative [Colmerauer 84]. En **programmation déclarative**, l'informaticien représente des *connaissances* sur un sujet donné. Ces connaissances seront exprimées sous forme d'une série de règles. Ces règles seront ensuite exploitées par la *machine Prolog*, capable de répondre à toute question dont la réponse est logiquement déductible des connaissances fournies au préalable.

III.1.2. La représentation des connaissances

Les expressions du langage PROLOG sont des *clauses de HORN* :

- une **clause** est une disjonction de littéraux positifs ou négatifs :
  $A_1 \vee A_2 \vee \ldots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_p$ avec $n,p > 0$
  Les B_i sont les conditions de la clause, les A_j en sont les conclusions.
- un **littéral positif** est de la forme $P(t_1\ t_2 \ldots t_m)$ où P est le nom d'un prédicat maître et les t_i sont des termes.
- un **terme** est défini récursivement. Les termes simples comprennent les constantes caractères ("mémoire", "prolog"), les constantes numériques, les constantes symboliques (":-") et les variables. Un terme structuré consiste en un foncteur (c'est-à-dire un symbole) d'arité n suivi de n termes.
- l'**arité** est un nombre qui correspond au nombre d'arguments d'un prédicat.

Une formulation équivalente des clauses peut être :

$$A_1 \vee A_2 \vee \ldots \vee A_n \leftarrow\!\!- B_1 \wedge B_2 \wedge \ldots \wedge B_p$$

La sémantique associée est la suivante :

- les variables sont quantifiées universellement,
- une clause est interprétée comme une implication logique classique

  $\forall (z_1\ z_2 \ldots z_l)\ A_1 \vee A_2 \vee \ldots \vee A_n \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_p$

  ce qui est équivalent à

  $\forall (x_1\ x_2 \ldots x_p)\ A_1 \vee A_2 \vee \ldots \vee A_n \leftarrow [\exists (y_1\ y_2 \ldots y_k)\ B_1 \wedge B_2 \wedge \ldots \wedge B_p]$

  où les x_i sont des variables apparaissant à la fois en condition et en conclusion, et les y_j, les variables apparaissant en condition mais pas en conclusion.

On démontre que la logique clausale est aussi puissante que la logique du premier ordre [Tärnlund 75].

Les clauses de Horn constituent un sous-ensemble de cette logique clausale :

- une **clause de Horn** est une clause ayant au plus un littéral positif.

Ainsi, une clause de Horn peut être de trois formes :

a) $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_p$ (on écrit $A \leftarrow B_1, B_2, \ldots, B_p$)

b) $A \leftarrow$ (on parlera alors d'*assertion*)

c) $\leftarrow B_1, B_2, \ldots, B_p$ (on parlera alors de *réfutation*)

Un **programme** en PROLOG est une suite de clauses de Horn de type a) ou b), le but recherché étant exprimé par une réfutation (clause de Horn de type c)). Ainsi, il n'y a pas de distinction entre la base de connaissances et la base de faits. Le but initial est repéré comme étant les conditions de la seule clause dont la partie conclusion est vide.

Par exemple, si nous savons que Socrate est un homme et que tous les hommes sont mortels, et que nous désirons savoir si Socrate est mortel, nous écrirons :

```
homme(socrate).          | Socrate est un homme
mortel(X) :- homme(X).   | X est mortel si X est un homme
```

Posons maintenant notre question :

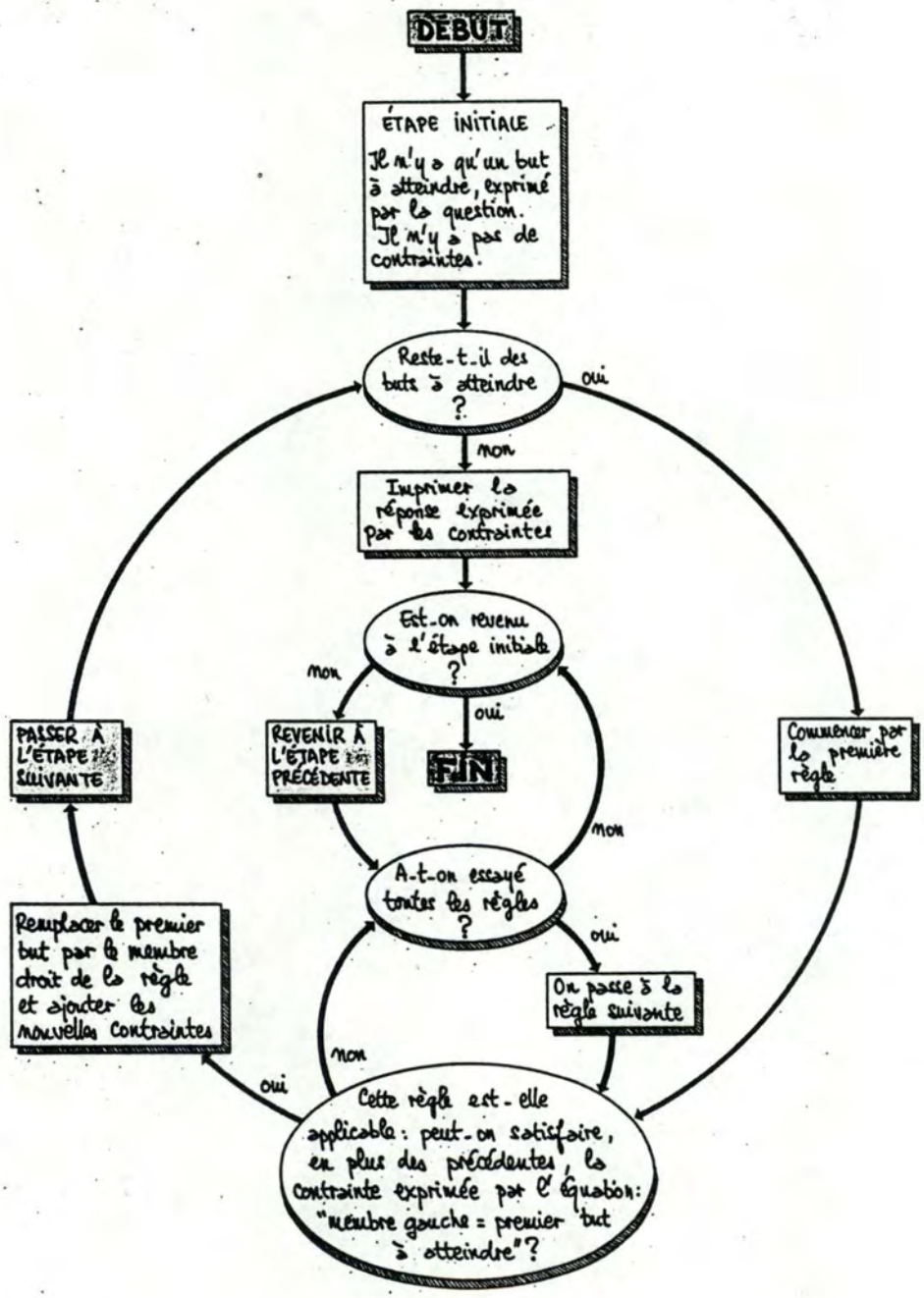
```
:- mortel(socrate).      | socrate est-il mortel
Vrai.                    | la machine nous répond
                         | affirmativement
```

Dans l'exemple précédent, *X* est une variable. L'utilisation des variables est la clé qui donne à Prolog sa puissance exceptionnelle, car elles permettent de raisonner sur des objets inconnus.

III.1.3. Fonctionnement d'un système Prolog

L'interprète du langage Prolog (ou machine Prolog) peut être considéré comme un moteur d'inférences procédant en chaînage-arrière, la base de connaissances étant constituée de clauses formant le programme. Le fonctionnement de la machine Prolog peut être schématisé comme suit :

LA MACHINE PROLOG (Colmerauer)



Au départ, il n'y a qu'une conjonction de buts formant **la question.**

Tant qu'il reste des buts à atteindre, on cherche à appliquer chacune des règles du programme. Prolog adopte une stratégie de recherche en profondeur d'abord. Lorsqu'une des clauses est applicable (c'est-à-dire que le but courant correspond - ou s'**unifie** - à la tête d'une règle), on l'applique, c'est-à-dire qu'on remplace le premier but de la conjonction à atteindre par le membre droit de la règle (éventuellement vide, on dit alors que le but est **effacé**), et on ajoute les contraintes correspondantes au début de l'ensemble des contraintes (ceci signifie que Prolog finit de satisfaire un sous-but avant d'essayer quoi que ce soit d'autre). Le parcours des règles se fait dans l'ordre de leur entrée dans le programme (et donc en commençant par les règles les plus anciennes). Si au cours de l'application d'une règle le système peut donner une valeur à une variable, on dira que cette variable est **instantiée**.

On applique récursivement la méthode de résolution exposée ci-dessus à la conjonction des buts obtenus, conjonction appelée **résolvante**.

Si à un moment donné le système ne parvient pas à prouver le premier but d'une résolvante, il revient à la précédente (ce phénomène est appelé **backtracking**) en rejetant la clause la plus récemment activée, et défait les instantiations correspondantes.

Il tente alors de résoudre le premier but de l'ancienne résolvante au moyen de clauses non encore essayées.

Les systèmes Prolog sont également non-déterministes, c'est-à-dire qu'il leur est possible, s'ils en sont requis, de fournir toutes les réponses à une question. Le non-déterminisme revient en fait à activer artificiellement le mécanisme de retour arrière après chaque réponse afin d'essayer systématiquement toutes les règles.

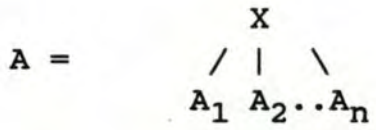
Cette implémentation particulière de Prolog peut amener le système à boucler.

III.2. Analyse de l'architecture de Prolog

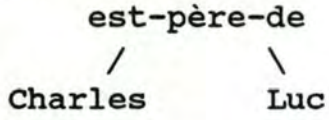
On ne trouvera dans cette partie que l'architecture élémentaire qu'on rencontre dans tout système Prolog. Pour des informations plus détaillées, nous renvoyons le lecteur à [Clantin 85].

III.2.1. Les structures de données du langage Prolog

La structure la plus générale des objets manipulés en Prolog, est la structure d'arbre. Un **arbre** A est constitué d'un noeud X appelé *racine* et d'un ensemble (éventuellement vide) ordonné d'éléments A_1, ..., A_n, qui sont eux-mêmes des arbres :



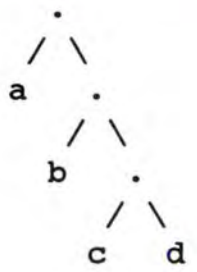
Par exemple, pour représenter que Charles est le père de Luc, on aura :



Tout noeud sans descendant est dit **terminal**.

En Prolog, un arbre peut être partiellement inconnu. Dans ce cas, l'un de ses noeuds terminaux au moins se réduit à une variable.

Une autre structure de données est la structure de liste. Une **liste** peut être construite grâce à l'opérateur infixé ".", pour lequel l'association se fait de droite à gauche. Ainsi, la liste a.b.c.d se représente par l'arbre :



L'efficacité de la structure de représentation des arbres dépend fortement de la manière de mémoriser les

identificateurs des symboles fonctionnels, constantes et prédicats. Ces derniers ont en effet une taille en caractères très variable et apparaissent généralement plusieurs fois dans le même programme.

III.2.2. Les structures de travail de Prolog

Typiquement, en Prolog, la mémoire de travail est divisée en trois parties : le stack, le heap et le trail.

III.2.2.1. Le Stack

L'ensemble de tous les buts et sous-buts de la résolution d'un programme Prolog peut être représenté sous la forme d'un arbre. En l'occurrence, il s'agit d'un arbre "*et*" qui stocke sous forme non redondante un chemin dans l'arbre de recherche depuis sa racine jusqu'à la résolvante courante. Cet arbre sera généralement implémenté sous forme de pile (stack en anglais). Chaque élément de la pile mémorisera un ensemble d'informations concernant une partie de l'arbre de résolution.

III.2.2.2. Le Heap

Cette partie de la mémoire est utilisée pour stocker les règles. Plusieurs techniques existent pour économiser l'espace mémoire. On peut citer le partage de structure et la recopie intermédiaire.

III.2.2.3. Le Trail

Dans cette portion de mémoire sont enregistrés les liens qui se font et se défont entre les variables référencées.

III.2.3. Le "garbage collector"

Au fur et à mesure du déroulement du programme, les piles grandissent. Leur organisation fait qu'on ne pourra récupérer leur espace de travail que lors d'un retour arrière. Il risque cependant d'arriver un moment où il n'y aura plus de place disponible. Il sera alors intéressant de pouvoir faire des récupérations anticipées. C'est dans ce but qu'ont été mises en oeuvre des techniques de récupération de mémoire [Bruynooghe 84].

III.3. Relevé des variables d'états

Une analyse de l'architecture de Prolog révèle un certain nombre de points qu'il est important de prendre en considération. On retiendra :

- la hauteur de la pile PROLOG (le stack),
- le nombre de fonctions de base et leur implémentation,
- le nombre de bibliothèques,
- la base de faits,
- la base de connaissances.

III.4. Des programmes de mesures

Nous nous sommes servis de plusieurs programmes pour tenter de cerner au mieux les performances de Prolog. Dans les pages qui suivent, nous allons présenter chacun des programmes et exposer brièvement leur utilité. Le lecteur intéressé pourra trouver à l'annexe 2 le code Prolog de ces benchmarks.

III.4.1. Inversion de listes

Cette procédure est un benchmark gros consommateur de listes. Il permet d'apprécier la manière dont l'implémentation du recopiage et du passage de ces listes a été réalisée. Ce benchmark effectue aussi un grand nombre de retours arrières et d'unifications.

III.4.2. Le "quick-sort"

On trouve ici un codage de l'algorithme classique de ce tri rapide appliqué aux listes. L'implémentation de ce tri donne une indication sur le comportement de Prolog en face de procédures effectuant de multiples sorties.

III.4.3. Transposition d'une liste de symboles en une liste de nombres

Ce programme sert à tester l'implémentation des variables logiques.

III.4.4. Interrogation d'une base de données

Ce programme teste la manière dont Prolog parcourt les prédicats.

III.4.5. Gestion dynamique d'une base de données

Ce programme permet d'analyser la manière dont Prolog gère dynamiquement son espace mémoire.

III.4.6. Le Sieve benchmark

Ce test calcule la quantité de nombre premiers jusqu'à une borne maximum.

Prolog a la réputation d'être destiné essentiellement à l'utilisation symbolique. Ce benchmark va donc permettre d'évaluer le comportement de Prolog pour le calcul avec des nombres entiers.

III.4.7. Le Float benchmark

Le même benchmark que le Sieve mais en utilisant uniquement des nombres réels.

III.5. Les produits testés

Notre choix (en fait limité aux versions dont nous avons pu disposer) s'est fixé sur trois implémentations. Par ordre alphabétique : micro-Prolog, Prolog-86 et Turbo Prolog.

On trouvera dans les lignes qui suivent une description sommaire de ces implémentations ainsi qu'une comparaison de leurs fonctionnalités.

III.5.1. micro-Prolog (version 3.1)

Nous devons cet interpréteur Prolog à la société Logic Programming Associates. Il s'agit d'un des premiers Prolog disponibles sur micro-ordinateur (après la version de l'équipe de Colmerauer sur Apple II). La première version de ce Prolog remonte à 1980 et était destinée aux micro-ordinateurs fonctionnant sous CP/M. Elle a été mise à jour et enrichie périodiquement pour aboutir au produit "mature" testé ici. La version actuelle est utilisable sur les machines fonctionnant sous CP/M et MS/DOS. Pour des raisons de compacité et d'efficacité, elle est en majeure partie écrite en assembleur. Ce produit ne sacrifie aucune des particularités des Prolog disponibles sur les gros ordinateurs.

La syntaxe standard de micro-Prolog est très rudimentaire et présente de grandes similitudes avec la syntaxe du langage Lisp. Le superviseur est lui aussi très simplifié. Il ne propose que les fonctions essentielles à la prise en charge des programmes et à leur exécution. Un gros avantage de ce superviseur réside dans ses possibilités d'extension. On trouve par exemple sur la disquette programme des extensions permettant de simuler le Prolog DECsystem-10 (cette possibilité n'est toutefois présente que sur les versions MS/DOS).

Les programmes micro-Prolog peuvent être structurés en modules. Un **module** est une collection de définitions de relations. Un module communique avec les autres programmes par l'intermédiaire de listes de noms déclarées spécialement pour cet import/export. Tous les autres noms déclarés dans le module sont internes à ce module. Un autre avantage des modules est qu'ils peuvent être chargés ou détruits individuellement.

Parmi les modules présents sur la disquette, on trouve par exemple un module d'édition, un module de trace et le module superviseur DECsystem-10.

Une des limitations du superviseur de base micro-Prolog est qu'il n'accepte que des noms de variables commençant par une des lettres X,Y,Z,x,y,z suivie éventuellement d'un nombre entier. Cette limitation nuit à la lisibilité des programmes rédigés en micro-Prolog standard. Cette limitation peut être contournée par une amélioration du superviseur, au prix toutefois d'une perte d'efficacité et d'une plus grande consommation de mémoire.

III.5.2. Prolog-86 (version 2.2)

Ce Prolog interprété de la société Expert Systems Ltd reprend un sous-ensemble du Prolog DECsystem-10. Cette implémentation est disponible pour toutes les machines fonctionnant sous MS/DOS.

Le programme est accompagné d'exemples portant sur la différenciation symbolique, les jeux, les systèmes-experts et la compréhension du langage naturel. L'éditeur est assez pauvre et orienté ligne.

III.5.3. Turbo Prolog (version 1.0)

Turbo Prolog est une version compilée de Prolog fonctionnant sur IBM PC et compatibles. Borland, la firme qui commercialise ce produit, a vraisemblablement conçu cette implémentation expressément pour les machines fonctionnant sous MS/DOS.

Cette implémentation est apparue sur le marché dès le deuxième quart de l'année 86. Le manuel de référence est complet et accompagné d'un excellent cours d'initiation. Un "gros" programme de démonstration, *Géobase*, accompagne le tout. Ce programme montre comment, grâce à Prolog, on peut interpréter le langage naturel anglais pour l'interrogation d'une base de données.

Turbo Prolog possède des prédicats d'entrées/sorties pour fichiers (accès séquentiels et directs), prend en compte les chaînes de caractères et possède des capacités sonores et graphiques ainsi que le multi-fenêtrage. De plus, il est possible, sans quitter le programme, de faire appel au système hôte.

Le temps de compilation est extrêmement bref. La compilation se passe en mémoire centrale. Le compilateur permet de générer aussi du code objet. Passer de l'exécution du programme compilé vers l'éditeur et vice-versa est quasi instantané. L'éditeur est du type pleine page multi-fenêtres.

Lors de la compilation, dès qu'une erreur est détectée, le compilateur s'arrête et l'éditeur est activé à l'endroit où l'erreur se trouve. Le compilateur possède des options permettant par exemple de détecter le déterminisme, de supprimer les *warnings*, de tracer un programme ou des prédicats, ...

La trace permet d'exécuter un programme pas à pas. Une fenêtre indique classiquement les clauses et les variables instantiées à un moment donné, mais, en plus, dans la fenêtre de l'éditeur, le curseur pointe sur les prédicats que le système parcourt tandis que dans une troisième fenêtre s'affichent les résultats normaux de l'exécution. Ce système est très puissant et très clair mais aussi très lent, car c'est l'utilisateur qui, en pressant une touche, fait progresser l'exécution.

Turbo Prolog semble être à la fois inspiré du "standard" exposé dans [Clocksin et al. 85] et du langage Pascal (le premier cheval de bataille de Borland). Comme Pascal, Turbo

Prolog est un langage typé dans lequel l'utilisateur peut définir ses propres types. Les types prédéfinis reprennent les entiers, les réels, les caractères, les chaînes, les fichiers et les symboles.

Tout comme Pascal, un programme Turbo Prolog est composé de plusieurs *sections*. Les définitions des types de variable se trouveront dans la section **domaine** (*domains*), les spécifications des types des arguments des prédicats dans la section **prédicats** (*predicates*), la définition des prédicats dans la section **clauses** (*clauses*), les spécifications des types des prédicats qu'on peut ajouter ou supprimer au moment de l'exécution dans une section **base de données** (*database*), ... Une section optionnelle appelée **but** (*goal*), permet de spécifier un ensemble de clauses qui seront résolues automatiquement à chaque exécution du programme.

Une des limitations du typage des données est que, si un prédicat est susceptible d'utiliser plusieurs types de données, il faut effectuer toutes les déclarations possibles dans la section prédicats. De plus, tous les prédicats portant le même nom doivent être d'arité identique.

III.5.4. Etude comparative des fonctionnalités

La présentation de ces fonctionnalités est inspirée de [Weeks et al. 86].

III.5.4.1. Présentation

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
Version :	3.1	2.2	1.0
Mémoire virtuelle	non	non	non
Modularisation	oui	non	non
Compilateur	non	non	oui
Accès au shell	non	non	oui
Editeur interne	oui	oui	oui
Nombre de prédicats	60	65	109

Remarques :

- l'absence de la mémoire virtuelle implique que l'espace de Prolog est limité par l'espace de la mémoire principale.
- la modularisation est une méthode qui permet d'isoler des procédures Prolog et d'éviter des conflits de noms. Cette possibilité permet à différents programmeurs de travailler indépendamment les uns des autres, en ne connaissant que les spécifications des autres modules.
- l'accès au shell signifie qu'il est possible de suspendre Prolog pour exécuter un autre programme puis de revenir à Prolog sans avoir altéré l'état dans lequel on se trouvait.

III.5.4.2. Prédicats prédéfinis

III.5.4.2.1. Prédicats d'entrée/sortie (I/O)

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
I/O programmes/clauses	2	4	2
I/O caractères	0	5	3
I/O chaînes de caractères	0	0	3
I/O termes	11	7	4
I/O flot/fichiers	6	8	17

Remarques :

- flot désigne le flot d'entrée courant.
- les prédicats d'entrée/sortie de caractères incluent les prédicats de sortie des caractères de contrôle qui permettent de passer à la ligne et à la tabulation suivante.

III.5.4.2.2. Prédicats d'entrée/sortie de programmes et de clauses

Abréviation :

- e.t. signifie espace de travail.

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
Sauve e.t. par prédicats	oui	non	non
Détruit e.t. par fichier	non	oui	oui
remplace e.t. avec fichier	non	oui	oui

Remarques :

- pour les prédicats d'entrée/sortie de programmes et clauses, les deux opérations de base sont : *charger un fichier dans l'espace de travail* et *sauver l'espace de travail dans un fichier*.
- dans [Clocksin et al. 85], les commandes de chargement de fichiers sont toujours cumulatives en regard de l'espace de travail. Prolog-86 dispose d'une variation qui efface d'abord l'espace de travail puis charge le fichier.

III.5.4.2.3. Prédicats d'entrée/sortie de caractères

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
reçu car. flot/fichier	non/non	oui/non	oui/oui
reçu sans écho de flot	non	non	oui
sauter au car. flot/fich	non/non	oui/non	oui/oui
idem sans écho de flot	non/non	non/non	oui/oui
envoyer car. à flot/fich	non/non	oui/non	oui/oui
nouvelle ligne flot/fich	non/non	oui/non	oui/oui
tabulation flot/fichier	non/non	oui/non	non/non

Remarque :

- en micro-Prolog, chaque caractère doit être lu ou écrit comme un terme.

III.5.4.2.4. Prédicats d'entré/sortie de chaînes de caractères

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
reçu chaîne flot/fichier	non/non	non/non	oui/oui
envoyer chaîne à flot	non	non	oui

III.5.4.2.5. Prédicats d'entrée/sortie de termes

Abréviations :

- l. signifie lire.
- é. signifie écrire.

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
l. terme depuis flot/fich	oui/oui	oui/non	oui/oui
l. symbole depuis flot/fich	non/non	oui/non	oui/oui
l. nombre depuis flot/fich	non/non	non/non	oui/oui
é. vers flot/fich	oui/oui	oui/non	oui/oui
l./é. formatée	non/non	non/non	non/oui
déclarer un opérateur	non	oui	non
définir un prompt	non	oui	non
accès direct fichier	oui	non	oui
l./é. en accès direct	oui	non	oui

III.5.4.2.6. Prédicats de contrôle

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
Contrôle flot/fichier	6	8	16
Succès/échec	2	2	2
Backtracking	1	2	4

III.5.4.2.7. Prédicats de contrôle de flot et de fichier

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
créer fichier	oui	non	oui
ouvrir flot/fichier	non/oui	oui/non	oui/oui
créer/ouvrir/fermer module	oui	non	non
rediriger flot entrée	non	oui	oui
rediriger flot sortie	non	non	oui
vérifier l'existence fich	non	non	oui
renommer un fichier	non	non	oui
détruire un fichier	oui	non	oui
lister le répertoire	oui	non	oui

III.5.4.2.8. Prédicats de succès/échec

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
true	non	oui	non
fail	oui	oui	oui

III.5.4.2.9. Prédicats de retour arrière

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
cut	oui	oui	oui
repeat	non	oui	non
opérateurs logiques	oui	oui	oui
call explicite	non	oui	non
nombre de solutions	oui	non	oui

Remarques :

- les opérateurs logiques reprennent : *and*, *or*, *not*.
- micro-Prolog possède un prédicat du type *if...then....*

III.5.4.2.10. Prédicats de classification des termes et de conversion

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
Total :	8	6	10
est une variable	oui	oui	oui
n'est pas une variable	non	oui	non
est un atome	non	oui	oui
est un nombre	oui	oui	oui
est un atome ou un nombre	oui	oui	non
est une liste	oui	non	non

III.5.4.2.11. Prédicats de comparaison de termes

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
Total :	4	8	7
matche et unifie	oui	oui	oui
ne matche pas	non	oui	non
équivalence	non	oui	oui
pas équivalent	non	oui	non
inégalités relationnelles	non	oui	oui

III.5.4.2.12. Prédicats d'évaluation arithmétique

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
Total :	5	1	1
évalue et unifie	non	oui	oui
entier le plus proche	oui	non	non

Remarque :

- en micro-Prolog, il n'y a d'opérateur que pour l'addition et la multiplication. Suivant les arguments qui sont évalués, on obtient l'addition ou la soustraction, la multiplication ou la division.

III.5.4.2.13. Opérateurs arithmétiques

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
int	non	non	non
float	non	non	non
ascii vers entier	non	non	oui
chaîne vers réel	non	non	oui
random	non	non	oui
modulo	non	oui	oui
division entière	non	non	oui
valeur absolue	non	non	oui
cosinus	non	non	oui
sinus	non	non	oui
tangente	non	non	oui
arc tangente	non	non	oui
exponentielle	non	non	oui
logarithme népérien	non	non	oui
logarithme en base 10	non	non	oui
racine carrée	non	non	oui
et logique	non	non	oui
non logique	non	non	oui
ou logique	non	non	oui
ou exclusif logique	non	non	oui
shift logique gauche	non	non	oui
shift logique droit	non	non	oui
appel à l'horloge	oui	non	oui
espace mémoire disponible	oui	oui	oui

III.5.4.2.14. Prédicats d'opérations de base de données

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
Contrôle de clause	7	6	3
Manipulation de structure	0	8	0

III.5.4.2.15. Prédicats de contrôle des clauses de base de données

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
lister toutes les clauses	oui	non	non
lister clauses spécifiées	oui	oui	non
ajouter une clause	oui	oui	oui
retirer :			
la première clause d'un prédicat	oui	oui	non
toutes les clauses d'un prédicat	oui	non	non

III.5.4.2.16. Prédicats de manipulation de structure

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
prédicat d'unif. de struct.	non	oui	non
donner le $n^{ième}$ argument	non	oui	non
conversion liste/structure	non	oui	non
conversion liste/atome	non	oui	non
longueur d'une liste	non	oui	non

III.5.4.2.17. Prédicats de chaînes de caractères

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
chercher une sous-chaîne	non	non	oui
diviser une chaîne	non	non	oui
longueur d'une chaîne	non	non	oui

III.5.4.2.18. Prédicats de debugging et de trace

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
Total :	6	3	4
trace d'un programme	oui	oui	oui
trace des buts	oui	oui	oui
rapport des buts tracés	oui	oui	oui

Remarque :

- les systèmes de trace offerts par Turbo Prolog fonctionnent automatiquement en mode pas à pas. Cela veut dire qu'à chaque tentative d'unification, le système s'arrête et demande à l'utilisateur d'appuyer sur une touche pour passer à la tentative d'unification suivante.
- les traces en Turbo Prolog sont très explicites. On dispose classiquement d'une fenêtre dans laquelle les résultats des unifications s'affichent mais en plus, dans la fenêtre d'édition, le curseur indique le prédicat que Prolog évalue.
- à chaque but, micro-Prolog s'arrête et demande à l'utilisateur s'il faut ou non tracer le but.

III.5.4.2.19. Prédicats divers

Turbo Prolog :

- offre des prédicats permettant de manipuler des fenêtres. L'utilisateur peut les créer, les déplacer, les supprimer, les superposer, leur donner des attributs (couleurs, taille, ...),
- permet d'accéder à toutes les possibilités graphiques de la machine. Il supporte même les nouvelles cartes haute définition,
- possède des prédicats permettant de gérer la sortie sonore du micro-ordinateur,
- permet d'accéder et de modifier n'importe quel byte de la mémoire,
- peut accéder aux ports d'entrée/sortie.

III.5.4.3. Limites des systèmes

Produit :	micro-Prolog	Prolog-86	Turbo Prolog
nombre de car. d'un nom	60		250
nombre de car. d'une constante chaîne			250
nombre de car. d'une variable chaîne			64k
domaine des entiers			2-bytes
codage des réels	4-bytes	8-bytes IEEE	8-bytes IEEE
nombre de paramètres dans un prédicat	64		50
nombre de nom de domaines	S.O.	S.O.	250
nombre d'alternatives dans une déclaration de dom.	S.O.	S.O.	250
nombre de prédicats différents			300
nombre de variable dans une clause			100
nombre de littéraux dans une clause			100
nombre de clauses dans chaque prédicat			500

Remarque :

- S.O. signifie *sans objet*.

III.6. Etude de l'instrumentation

III.6.1. Point de vue temps CPU

Les tests ont été effectués sur un I.B.M. PC équipé d'une horloge interne. Cette horloge est précise au 1/52^{ième} de seconde.

Turbo Prolog et micro-Prolog offrent des fonctions d'accès à l'horloge interne. Une lecture de l'horloge est effectuée au début et à la fin de chaque test.

Pour Prolog-86, le chronométrage à été fait manuellement.

III.6.2. Point de vue espace mémoire

Nous nous sommes seulement intéressés à la mémoire primaire. Tous les Prolog testés possèdent des prédicats renseignant l'utilisateur sur cet espace mémoire.

Turbo Prolog possède un prédicat renvoyant le nombre d'unités de stack, de heap et de trail utilisés.

Prolog-86 se contente d'indiquer le nombre de bytes utilisés.

micro-Prolog, le moins bien loti des trois, donne le nombre de kilo-bytes encore disponibles.

III.7. Les jeux de tests

Des premières mesures ont été faites en exécutant une fois chaque programme. Ces mesures donnent une indication du comportement du système dans des conditions quasi *optimales* puisque, au départ du test, la mémoire est vide.

Nous avons ensuite voulu voir comment se comportaient les différents Prolog lorsque la mémoire commençait à se remplir. Pour cela, nous avons simulé une boucle *for* afin d'exécuter plusieurs fois de suite le même programme. En général, chaque programme a ainsi été exécuté 1, 2, 4, 8 et 16 fois, dans les limites permises par l'espace mémoire.

III.8. Les mesures

Les résultats des mesures sont exposés suivant le temps CPU et l'espace mémoire.

Pour quelques-uns des tests, nous avons indiqué les temps mesurés par [Warren 77] pour le Prolog compilé DECsystem-10.

Les temps sont exprimés en secondes et dixièmes de seconde.
L'espace mémoire utilisé est exprimé :

- pour micro-Prolog : en kilo-octets,
- pour Prolog-86 : en octets,
- pour Turbo Prolog : en octets,
- pour la colonne total : en octets.

Un trait horizontal "-" indique que le test n'a pas pu être réalisé.

Explication des abréviations :

- it : le nombre d'itérations. L'itération 0 correspond à un appel direct du programme, sans passer par la boucle *for*,
- l.i. : représente le nombre d'inférences logiques,
- lips : représente le nombre d'inférences logiques par seconde et

$$\text{le nombre de lips} = \frac{\text{l.i.}}{\text{temps}},$$

- t : trail.

III.8.1. Inversion de listes

III.8.1.1. Point de vue temps CPU

it	l.i.	micro-Prolog		Prolog-86		Turbo Prolog	
##	#####	temps	lips	temps	lips	temps	lips
0	466	2"3	**202**	2"7	**172**	0"1	**4660**
1	468	2"3	**203**	3"2	**145**	0"1	**4670**
2	936	4"6	**203**	5"3	**176**	0"3	**2335**
4	1872	-	-	10"0	**187**	0"5	**3120**
8	18720	-	-	-	-	4"3	**4653**
16	34440	-	-	-	-	8"7	**4303**
Lips moyen			**203**		**170**		**3957**
écart type			1		18		994

A titre indicatif, une exécution du programme prend 53,7 millisecondes en DECsystem-10 Prolog, ce qui représente environ 8641 LIPS.

III.8.1.2. Point de vue espace mémoire

it	micro-Prolog		Prolog-86			Turbo Prolog			
##	kb	total	local	global	total	stack	heap	t	total
0	20	**20480**	7584	3700	**11284**	52	3049	0	**3101**
1	20	**20480**	7848	3696	**11544**	52	3049	0	**3101**
2	39	**30720**	15694	7242	**22936**	52	6098	0	**6150**
4	-	-	31386	14334	**45720**	52	12196	0	**12248**
40	-	-	-	-	-	52	121960	0	**122012**
80	-	-	-	-	-	52	243920	0	**243972**

III.8.2. Le "quick-sort"

III.8.2.1. Point de vue CPU

it	l.i.	micro-Prolog		Prolog-86		Turbo Prolog	
##	#####	temps	lips	temps	lips	temps	lips
0	499	3"5	**143**	7"1	**70**	0"2	**2495**
1	501	3"5	**143**	7"8	**64**	0"3	**1670**
2	1002	7"6	**132**	14"5	**69**	0"5	**2004**
4	2004	15"4	**130**	28"2	**71**	0"9	**2227**
8	4008	-	-	-	-	1"7	**2358**
16	8016	-	-	-	-	3"3	**2429**
Lips moyen		**137**		**69**		**2197**	
écart type		7		3		311	

A titre indicatif, une exécution du programme prend 75,0 millisecondes en DECsystem-10 Prolog, ce qui représente environ 6653 LIPS.

III.8.2.2. Point de vue espace mémoire

it	micro-Prolog		Prolog-86			Turbo Prolog			
##	kb	total	local	global	total	stack	heap	t	total
0	11	**11264**	5144	2546	**7690**	52	3854	0	**3906**
1	11	**11264**	5138	2542	**7680**	52	3854	0	**3906**
2	22	**22528**	10274	4934	**15208**	52	7708	0	**7760**
4	-	-	20546	9178	**29724**	52	15416	0	**15468**
8	-	-	-	-	-	52	30832	0	**30884**
16	-	-	-	-	-	52	61664	0	**61716**

III.8.3. Transposition d'une liste de symboles en une liste de nombres

III.8.3.1. Point de vue temps CPU

it	l.i.	micro-Prolog		Prolog-86		Turbo Prolog	
##	#####	temps	lips	temps	lips	temps	lips
0	263	2"0	**132**	2"4	**110**	-	-
1	265	2"0	**133**	2"9	**91**	-	-
2	530	4"0	**133**	5"0	**106**	-	-
4	1060	8"5	**125**	9"2	**115**	-	-
8	2120	17"2	**123**	17"3	**123**	-	-
16	4240	-	-	34"0	**125**	-	-
Lips moyen		**129**		**112**		-	
écart type		5		12		-	

A titre indicatif, une exécution du programme prend 40,2 millisecondes en DECsystem-10 Prolog, ce qui représente environ 6542 LIPS.

III.8.3.2. Point de vue espace mémoire

it	micro-Prolog		Prolog-86			Turbo Prolog			
##	kb	total	local	global	total	stack	heap	t	total
0	4	**4096**	932	1380	**2312**	-	-	-	-
1	4	**4096**	926	1376	**2302**	-	-	-	-
2	8	**8192**	1850	2602	**4452**	-	-	-	-
4	15	**15360**	3698	5054	**8752**	-	-	-	-
8	30	**30720**	7394	9958	**17352**	-	-	-	-
16	-	-	14786	19766	**34552**	-	-	-	-

III.8.4. Interrogation d'une base de données

III.8.4.1. Point de vue temps CPU

it	l.i.	micro-Prolog		Prolog-86		Turbo Prolog	
##	#####	temps	lips	temps	lips	temps	lips
0	236	64"3	3	70"2	3	4"7	50
1	238	64"3	3	70"3	3	4"8	50
2	476	128"5	3	139"6	3	9"5	50
4	952	257"2	3	278"0	3	18"9	50
8	1904	514"7	3	554"8	3	37"8	50
16	3808	1029"0	3	1108"5	3	75"6	50
Lips moyen			3		3		50
écart type			0		0		0

A titre indicatif, une exécution du programme prend 185 millisecondes en DECsystem-10 Prolog, ce qui représente environ 1276 LIPS.

III.8.4.2. Point de vue espace mémoire

it	micro-Prolog	Prolog-86				Turbo Prolog			
##	kb	total	local	global	total	stack	heap	t	total
0	0	**0**	14	162	**176**	52	0	0	**52**
1	0	**0**	14	158	**172**	52	0	0	**52**
2	0	**0**	26	164	**190**	52	0	0	**52**
4	0	**0**	50	176	**226**	52	0	0	**52**
8	0	**0**	98	200	**298**	52	0	0	**52**
16	0	**0**	194	248	**442**	52	0	0	**52**

III.8.5. Gestion d'une base de données

III.8.5.1. Point de vue temps CPU

it	l.i.	micro-Prolog		Prolog-86		Turbo Prolog	
##	#####	temps	lips	temps	lips	temps	lips
0	104	6"8	**15**	15"7	**7**	0"8	**130**
1	106	6"9	**15**	15"8	**7**	0"8	**133**
2	212	13"8	**15**	48"8	**4**	1"5	**141**
4	424	27"7	**15**	173"4	**2**	3"0	**141**
8	848	55"5	**15**	665"4	**1**	5"7	**149**
16	1696	111"2	**15**	*2472"6	**1**	11"4	**149**
Lips moyen		**15**		**4**		**141**	
écart type		0		3		8	

III.8.5.2. Point de vue espace mémoire

it	*micro-Prolog*	Prolog-86			Turbo Prolog			
##	kb	total	local	global	total	stack	heap	t	total
0	0	**0**	14	162	**176**	0	1400	0	**1400**
1	0	**0**	14	158	**172**	52	1400	0	**1452**
2	0	**0**	26	164	**190**	52	1400	0	**1452**
4	0	**0**	50	176	**226**	52	1400	0	**1452**
8	0	**0**	98	200	**298**	52	1400	0	**1452**
16	0	**0**	194	248	**442**	52	1400	0	**1452**

III.8.6. Le Sieve benchmark

III.8.6.1. Point de vue temps CPU

it	l.i.	micro-Prolog		Prolog-86		Turbo Prolog	
##	#####	temps	lips	temps	lips	temps	lips
0	78	2"2	**35**	2"7	**29**	0"1	**780**
1	80	2"3	**35**	2"8	**29**	0"1	**800**
2	160	4"7	**34**	4"6	**35**	0"2	**800**
4	320	14"8	**22**	9"6	**33**	0"4	**800**
8	640	52"1	**12**	25"4	**25**	0"9	**711**
16	1280	207"1	**6**	82"0	**16**	2"8	**457**
Lips moyen			**24**		**28**		**725**
écart type			13		7		136

III.8.6.2. Point de vue espace mémoire

it	micro-Prolog	Prolog-86			Turbo Prolog			
##	kb	total	local	global	total	stack	heap	t	total
0	1	**1024**	832	582	**1414**	872	196	0	**1068**
1	1	**1024**	826	572	**1398**	906	196	0	**1102**
2	2	**2048**	1660	1010	**2670**	1790	350	0	**2140**
4	5	**5120**	3328	1874	**5202**	3558	658	0	**4216**
8	10	**10240**	6664	3602	**10266**	7094	1274	0	**8368**
16	19	**19456**	13336	7058	**20394**	14166	2506	0	**16672**

III.8.7. Le Float benchmark

III.8.7.1. Point de vue temps CPU

it	l.i.	micro-Prolog		Prolog-86		Turbo Prolog	
##	#####	temps	lips	temps	lips	temps	lips
0	78	2"2	**35**	3"0	**26**	0"2	**390**
1	80	2"2	**36**	3"0	**27**	0"2	**400**
2	160	4"7	**34**	5"3	**30**	0"4	**400**
4	320	14"7	**22**	11"4	**28**	0"8	**400**
8	640	52"1	**12**	30"4	**21**	2"3	**278**
16	1280	207"1	**6**	99"8	**13**	6"9	**186**
Lips moyen			**24**		**24**		**342**
écart type			13		6		90

III.8.7.2. Point de vue espace mémoire

it	micro-Prolog	Prolog-86			Turbo Prolog			
##	kb	total	local	global	total	stack	heap	t	total
0	1	**1024**	832	1130	**1962**	1490	313	0	**1803**
1	1	**1024**	826	1126	**1952**	1542	313	0	**1855**
2	2	**2048**	1660	2106	**3766**	3068	575	0	**3643**
4	5	**5120**	3328	4066	**7394**	6120	1099	0	**7219**
8	10	**10240**	6664	7986	**14650**	12224	2147	0	**14371**
16	19	**19456**	13336	15826	**29162**	24432	4243	0	**28675**

III.9. Interprétation des résultats

micro-Prolog est un produit relativement complet. Il ne lui manque que quelques fonctions arithmétiques pour être parfait. Si le superviseur de base est assez pauvre, la facilité avec laquelle on peut l'améliorer compense ce défaut au prix, il est vrai, d'une augmentation de l'ordre de 30 pour cent du temps de réponse.

La possibilité de travailler avec des modules est appréciable pour développer du code de manière autonome, sans se soucier de la manière dont est implémentée le reste de l'application.

Prolog-86 est aussi complet que micro-Prolog. Il s'agit d'un excellent produit d'initiation car il suit fidèlement la "norme" de [Clocksin et al 85].

Le produit est cependant peu convivial. Il est impossible, par exemple, de lister toutes les clauses et l'éditeur est assez faible.

Turbo Prolog est un produit performant. Les temps de compilation sont très brefs et l'exécution est très rapide. Ce produit possède pourtant quelques limitations sérieuses comme le typage de ses données, le fait que les virgules ne sont pas des opérateurs au sens de [Cloksin et al. 85], le prédicat *call* n'existe pas, l'opérateur = fonctionne plutôt comme l'opérateur *is* de [Cloksin et al. 85] et on ne peut pas déclarer de nouveaux opérateurs. Ces limitations font que Turbo Prolog semble plutôt destiné aux programmeurs qui ont fait leurs classes avec des langages conventionnels et qui désirent toucher à l'intelligence artificielle.

Le produit souffre également de certaines erreurs de jeunesse. Par exemple, la directive *diagnostics* du compilateur ne fonctionne pas, ou encore, on constate un comportement bizarre dans le programme du quicksort.

III.10. Conclusions

Cette étude fait ressortir le fait que l'on ne peut juger un produit uniquement sur ses performances. Si par exemple Turbo Prolog est le plus rapide, ce n'est pas pour autant le choix le plus judicieux pour n'importe quelle application. Par exemple, pour réécrire un interpréteur Prolog en Prolog, il suffit en Prolog-86 de 4 prédicats alors qu'en Turbo Prolog, il en faut plus d'une cinquantaine. Ceci est dû au fait que les variables Turbo Prolog sont typées et qu'il n'est pas possible de réaliser un appel explicite à un prédicat.

Cette étude montre indirectement que les Prolog disponibles sur micro-ordinateurs arrivent à maturité.

CHAPITRE IV : D'autres études

IV.1. OPS5

IV.1.1. Introduction

OPS (*Official Problem Solver*) est un système-expert vide utilisant les règles de production. Il est développé à la Carnegie Mellon University par C. Forgy et son équipe depuis 1975. OPS est disponible sous plusieurs versions dont les plus récentes sont OPS4, OPS5, OPS 83 et OPS5+, cette dernière étant une adaptation de OPS au micro-ordinateur d'IBM.

Les systèmes de production sont très largement utilisés en IA dans la modélisation du comportement intelligent et la construction de systèmes-experts. C. Forgy a présenté dans sa thèse de doctorat [Forgy 79] quelques mesures concernant les caractéristiques de structure et de fonctionnement de grands programmes utilisant les règles de production. Depuis lors, les champs d'application des systèmes de production se sont largement étendus. C. Forgy et son équipe ont récemment procédé à de nouvelles mesures de certaines caractéristiques de différents systèmes-experts opérationnels utilisant OPS 5, afin de pouvoir chiffrer les gains correspondant à certaines optimisations et à une architecture parallèle [Gupta et al. 83].

Les caractéristiques mesurées sont aussi bien statiques (ce sont les données disponibles sans faire tourner le programme) que dynamiques.

Les six produits mesurés sont :

- R1, un programme pour la configuration de systèmes d'ordinateurs VAX (1932 règles),
- XSEL, un programme d'aide à la vente de VAX (1443 règles),
- PTRANS, un programme pour la gestion de production (1016 règles),
- HAUNT, un jeu d'aventures (834 règles),
- DAA, un programme de conception de circuits VLSI (131 règles),

- SOAR, un programme expérimental de résolution de problèmes (103 règles).

Les mesures portent entre autres sur le nombre de règles, de conditions par règle, de conclusions par règle, de variables par règle et d'occurrences de variables par condition; puis sur les répartitions entre les différents types de conclusions (*make* affirme un fait, *modify* le modifie, *remove* le détruit, *write* provoque une impression, ...). On trouve ensuite les caractéristiques dynamiques et les chiffres moyens et maximaux de la base de faits, de l'ensemble de conflits, de la base de connaissances et du nombre de modifications de faits.

IV.1.2. Quelques données chiffrées

système	R1	XSEL	PTRAN	HAUNT	DAA	SOAR
nombre de règles	1932	1443	1016	834	131	103
conditions par règle	5,58	3,84	3,12	2,41	3,91	5,80
conclusions par règle	2,90	2,41	3,64	2,51	2,86	1,83
variables par règle	9,00	3,68	6,68	0,16	10,5	9,86
var. par partie condition	1,76	1,66	1,67	1,39	1,40	2,13
conclusions *make* (en %)	34	31	22	10	34	71
conclusions *modify* (en %)	25	35	15	19	7	12
conclusions *remove* (en %)	13	8	7	25	16	0
conclusions *write* (en %)	9	3	10	44	17	17
autres conclusions (en %)	17	20	44	2	13	7
base de faits moyenne	-	62	-	60	708	353
base de faits maximale	-	89	-	63	1191	-
ensemble de conflits moyen	-	10	-	2	43	9
ensemble de conflits maxi.	-	22	-	6	421	26
base de connaissances moy.	-	368	-	473	1904	2413
base de connaissances max.	-	559	-	487	4616	-
modifications de faits	1247	756	984	559	16839	631

IV.2. Les modes représentation des connaissances

IV.2.1. Introduction

Nous avons vu dans le chapitre premier qu'il existait plusieurs modes de représentation des connaissances : les systèmes de production, les frames et les systèmes logiques. Dès lors, si nous disposons de ces trois modes de représentation des connaissances, lequel choisir pour résoudre un problème ?

Pour nous aider à choisir, Niwa et son équipe [Niwa et al. 84] ont essayé de résoudre un même problème avec chacun des modes de représentation cités ci-dessus avec en plus les systèmes de production structurés (dans ces systèmes, les connaissances sont structurées suivant leur ordre d'apparition lors de la résolution). Le but de leur étude était de comparer les difficultés d'implémentation de la base de connaissances et du moteur d'inférences, et les performances lors de l'exécution. Ils ont développé ces systèmes en Franz-Lisp sur un VAX 11/780.

Le problème que les systèmes-experts devaient résoudre concernait la gestion du risque dans la construction de grands projets informatiques.

Chaque système est constitué d'une base de connaissances, d'un moteur d'inférences et d'un système de mise à jour des connaissances. Le moteur d'inférences peut fonctionner en chaînage-avant et en chaînage-arrière. Dans ce dernier mode, il est capable d'expliquer son raisonnement. Les fonctions du système de mise à jour des connaissances permettent à l'utilisateur d'ajouter, de remplacer ou de supprimer facilement des connaissances.

IV.2.2. Quelques données chiffrées

Légende :

(1) système de production,
(2) système de production structuré,
(3) système de frame,
(4) système logique.

système	(1)	(2)	(3)	(4)
nombre de règles	263	263	213	348
taille de la base de connaissances	15	15	29	17
taille du moteur d'inférences	9,1	9,5	14,3	11,3
temps en chaînage-avant pour :				
50 clauses	0,3	0,2	0,2	1,0
100 clauses	0,5	0,3	0,2	2,0
200 clauses	0,8	0,4	0,2	4,0
temps en chaînage-arrière pour :				
50 clauses	0,5	0,3	0,3	0,7
100 clauses	0,6	0,3	0,3	1,2
200 clauses	0,8	0,3	0,3	2,0

Remarques :

- les tailles sont données en kilo-octets,
- les temps sont exprimés en secondes.

CONCLUSIONS

Nous avons passé en revue quelques aspects performances des systèmes-experts. Cette étude n'est en aucun cas une analyse exhaustive de ces systèmes du point de vue performance. Une analyse des aspects statique et dynamique d'un produit, quel qu'il soit, ne donnera jamais que des **indices** de comportement de ce produit en face de situations réelles.

On peut imaginer des programmes étalons et des protocoles de mesure pour les systèmes-experts, mais ces programmes de mesure seront difficilement *portables* d'un produit vers un autre. Il semble impossible, à partir d'un programme étalon type, de pouvoir mesurer les performances de tous les systèmes-experts existants.

Des standards de mesure, comme le LIPS, fournissent des indications sur l'aspect performance d'un produit. Ces indications ne sont cependant valables que si elles sont accompagnées des programmes et des protocoles de mesure qui ont servi à les établir.

Il faut aussi tenir compte du langage dans lequel a été rédigé le système-expert et de la machine sur laquelle il fonctionne. En effet, ces paramètres seront implicitement pris en compte par d'éventuels programmes de mesure.

Les systèmes-experts sont en plein essort. Le nombre de systèmes généraux double quasiment chaque année. Il apparaît important d'établir des *standards* de mesures afin d'aider les utilisateurs dans le choix d'un système.

Bibliographie

[Albert et al. 85] P. ALBERT et F.X. DELASSUS : *Knowledge Representation Object Oriented Language : manuel de référence.* Bull, 1985.

[Borland 86] BORLAND : *Turbo Prolog owner's handbook.* Borland International, 1986.

[Bowen 79] K.A. BOWEN : *Prolog.* School of Computer and Information Science, Syracuse University, September 1979.

[Bowen et al. 82] D.L. BOWEN, L. BYRD, F.C.N. PEREIRA, L.M. PEREIRA, D.H.D. WARREN : *DECsystem-10 Prolog user's manual.* Department of Artificial Intelligence, University of Edinburgh, November 1982.

[Bruynooghe 82] M. BRUYNOOGHE : *The memory management of Prolog implementations.* Logic Programming, pp. 83-96, Academic Press, London, 1982.

[Bruynooghe 84] M. BRUYNOOGHE : *Garbage collection in Prolog interpreters.* Implementations of Prolog, J.A. Campbell editor, pp. 259-267, New-York, 1984.

[Chailloux 85] J. CHAILLOUX : *Le_Lisp de l'INRIA : Manuel de référence.* I.N.R.I.A., Janvier 1985.

[Clantin 85] M. CLANTIN : *Techniques d'implémentation du langage Prolog.* Thèse de fin d'études en Licence et Maîtrise en Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, 1985.

[Clark et al. 85] K.L. CLARK et F.G. McCABE : *micro-Prolog : Programmer en logique.* Eyrolles, 1985.

[Clocksin et al. 85] W.F. CLOCKSIN et C.S. MELLISH : *Programmer en Prolog.* Editions Eyrolles, Avril 85.

[Cohen 85] J. COHEN : *Describing Prolog by its interpretation and compilation.* Communications of the ACM, Vol. 28, No. 12, pp. 1311-1324, December 1985.

[Colmerauer 82] A. COLMERAUER : *Prolog and infinite trees.* Logic Programming, pp. 231-251, Academic Press, London, 1982.

[Colmerauer 84] A. COLMERAUER : *Prolog, langage de l'intelligence artificielle.* La Recherche, No. 158, pp. 1104-1114, Septembre 1984.

[Colmerauer 85] A. COLMERAUER : *Prolog in 10 figures.* Communications of the ACM, Vol. 28, No. 12, pp. 1296-1310, December 1985.

[Colmerauer et al. 83] A. COLMERAUER, H. KANOUI, M. VAN CANEGHEM : *Prolog, bases théoriques et développements actuels.* Technique et Science Informatiques, Vol. 2, No. 4, pp. 271-311, 1983.

[Cordier 84] M.O. CORDIER : *Les systèmes experts.* La Recherche, No. 151, pp. 60-70, Janvier 1984.

[Cordier et al. 85] M.O. CORDIER, B. FALLER, M.C. ROUSSET : *Optimisation de l'opération de "pattern-matching" dans les systèmes experts.* LRI, Université Paris-Sud, Orsay, 1985.

[Cullingford et al. 83] R.E. CULLINGFORD and L.J. JOSEPH : *A heuristically 'optimal' knowledge base organisation technique.* Automatica, Vol. 19, No. 6, pp. 647-654, 1983.

[Faller 85] B. FALLER : *Traitement de connaissances incomplètes sur une application nécessitant un "pattern matching" efficace : le jeu de la carte au Bridge.* Thèse de 3^{e} cycle, Université de Paris-Sud, Paris, Juin 1985.

[Farreny 85] H. FARRENY : *Les systèmes experts.* Cepadues-Editions, Juillet 1985.

[Fikes et al. 85] R. FIKES, G.D. FINN, M.R. GENESERETH, M.L. GINSBERG, J. GUTKNECHT, H. HAMBURGER, F. HAYES-ROTH, T. KEHLER, H. SAMET, J.R. SLAGLE : *Special section on architectures for knowledge-based systems.* Communications of the ACM, Vol. 28, No. 9, pp. 904-1004, September 1985.

[Forgy 79] C.L. FORGY : *On the efficient implementations of production systems.* PhD thesis, Carnegie-Mellon University, 1979.

[Fuchi 85] K. FUCHI : *Ordinateurs dans les brumes.* Sciences & Avenir, numéro spécial "Les Japonais", No. 55, pp. 82-85, 1985.

[Gallaire 81] H. GALLAIRE : *Le langage Prolog.* Support de cours d'ingéniorat en intelligence artificielle, reconnaissance des formes et robotique, Toulouse, 1981.

[Gallaire 85] H. GALLAIRE : *La représentation des connaissances.* La Recherche, No. 170, pp. 1240-1248, Octobre 1985.

[Greussay 77] P. GREUSSAY : *Contribution à la définition interprétative et à l'implémentation des lambda-langages.* Thèse, Université de Paris VII, 1977.

[Guillemaud 84] JJ. GUILLEMAUD : *Puissance informatique : idées fausses et vrais problèmes.* Correspondance intérieure BULL, Octobre 1984.

[Gupta et al. 83] A. GUPTA and C.L. FORGY : *Measurements on production systems.* Department of Computer Science, Carnegie-Mellon University, December 1983.

[Hammond et al. 85] P. HAMMOND, F. KRIWACZEK, M. SERGOT : *Logic programming for expert systems.* Professorship in International Computer Science, Expert Systems, Mons, April-May 1985.

[Heering et al. 86] J. HEERING and P. KLINT : *The efficiency of the equation interpreter compared with the UNH Prolog interpreter.* SIGPLAN Notices, Vol. 21, No. 2, pp. 18-21, February 1986.

[Jackson 85] P. JACKSON : *Knowledge representation.* Department of Artificial Intelligence, University of Edinburgh, 1985.

[Kowalski 79] R. KOWALSKI : *Algorithm = Logic + Control.* Communications of the ACM, Vol. 22, No. 7, pp. 424-436, July 1979.

[Kowalski 79] R. KOWALSKI : *Logic for problem solving.* Artificial intelligence series, North Holland, 1979.

[Kowalski 82] R. KOWALSKI : *Logic as a computer language.* Logic Programming, pp. 3-16, Academic Press, London, 1982.

[Kowalski 85] R. KOWALSKI : *Logic Programming.* BYTE, pp. 161-177, August 1985.

[Lacroix 85] V. LACROIX : *Knowledges Oriented Object Language, adaptation au micro-ordinateur Micral 30.* CII-Honywell-Bull, Août 1985.

[Launet 85] E. Launet : *Le « génie de la connaissance » continue de souffler.* 01Informatique, Novembre 1985.

[Laurent 84] J.P. LAURENT : *La structure de contrôle dans les systèmes experts.* Technique et Science Informatiques, Vol. 3, No. 3, pp. 161-177, 1984.

[Lindstrom 79] G. LINDSTROM : *Backtracking in a generalized control setting.* ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, pp. 8-26, July 1979.

[Martelli et al. 82] A. MARTELLI and U. MONTANARI : *An efficient unification algorithm.* ACM Transactions on Programming Languages and Systems, Vol. 4, No. 2, pp. 258-282, April 1982.

[McCabe et al. 84] F.G. McCABE, K.L. CLARK, B.D. STEEL : *micro-PROLOG 3.1 programmer's reference manual.* Logic Programming Associates Ltd, March 1984.

[McDermott et al. 78] J. McDERMOTT, A. NEWELL, J. MOORE : *The efficiency of certain production system implementations.* Logic Programming, pp. 155-176, Academic Press, 1978.

[McDermott et al. 78] J. McDERMOTT and C. FORGY : *Production system conflict resolution strategies.* Logic Programming, pp. 177-199, Academic Press, 1978.

[Mellish 82] C.S. MELLISH : *An alternative to structure sharing in the implementation of a Prolog interpreter.* Logic Programming, pp. 99-101, Academic Press, London, 1982.

[Meyer 84] B. MEYER : *On formalism in specifications.* Department of Computer Science, University of California, Santa Barbara, June 1984.

[Micro-AI 84] MICRO-AI : *Prolog-86 user's guide and reference Manual.* Micro-AI, Rheem Valley, 1984.

[Niwa et al. 84] K. NIWA, K. SASAKI, H. IHARA : *An experimental comparison of knowledge representation schemes.* The AI Magazine, pp. 29-36, Summer 1984.

[Noirhomme et al. 84] M. NOIRHOMME et J. RAMAEKERS : *Performances et mesures.* Support de cours de Licence et Maîtrise en Informatique, Facultés Notre-Dame de la Paix, Namur, 1984.

[Paans 85] R. PAANS : *A close look at MVS systems: mechanisms, modes of use and security.* Thèse de doctorat, Delft University of Technology, December 1985.

[Queinnec 84] C. QUEINNEC : *Lisp mode d'emploi.* Eyrolles, Juillet 1984.

[Robinson 68] J.A. ROBINSON : *The generalized resolution principle.* Machine Intelligence 3, pp. 77-94, Edinburgh University Press, 1968.

[Roche 84] C. ROCHE : *LRO, Langage de Représentation d'Objet.* Rapport Bigre, No. 41, pp. 101-119, Octobre 1984.

[Rousset 83] M.C. ROUSSET : *TANGO, moteur d'inférences pour une classe de systèmes-experts avec variables.* Thèse de 3^{e} cycle, Université de Paris-Sud, Paris, Octobre 1983.

[Rubin 86] D. RUBBIN : *Inside Turbo Prolog.* Computer Language, Vol. 3, No. 7, pp. 23-28, July 1986.

[Sansonnet 85] J-P. SANSONNET : *Les machines de l'intelligence artificielle.* La Recherche, No. 170, pp. 1228-1239, Octobre 1985.

[Stefik et al. 82] M. STEFIK, J. AIKINS, R. BALZER, J. BENOIT, L. BIRNBAUM, F. HAYES-ROTH, E. SACERDOTI : *The organization of expert systems, a tutorial.* Artificial Intelligence, Vol. 18, pp. 135-173, 1982.

[Tärnlund 75] S.A. TARNLUND : *Logic information processing.* University of Stockolm, 1975.

[van Lamsweerde 85] A. VAN LAMSWEERDE : *Techniques d'Intelligence Artificielle.* Support de cours de Licence et Maîtrise en Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, 1985.

[Warren 77] D.H.D. WARREN : *Implementing Prolog - compiling predicate logic programs.* Research Reports 39 & 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[Warren 79] D.H.D. WARREN : *Prolog on the DECsystem-10.* Expert systems in the micro-electronic age, pp. 112-121, Editions Michie, 1979.

[Warren 80] D.H.D. WARREN : *An improved Prolog implementation which optimises tail recursion.* Logic Programming Workshop, 1980.

[Warren et al. 77] D.H.D. WARREN, L.M. PEREIRA, F.C.N. PEREIRA : *Prolog - the language and its implementation compared with Lisp.* SIGPLAN Notices, Vol. 12, No. 8 (special issue), pp. 109-115, 1977.

[Weeks et al. 86] J. WEEKS and H. BERGHEL : *A comparative feature-analysis of microcomputer Prolog implementations.* SIGPLAN Notices, Vol. 21, No. 2, pp. 46-61, February 1986.

[Weiss et al. 84] S. WEISS and G. VULIKOWSKI : *A practical guide to designing expert systems.* Bowman Allanheld Publishers, 1984.

[Winston 77] P. WINSTON : *Artificial Intelligence.* Addison-Wesley, 1977.

ANNEXE 1 : Le programme de génération automatique d'environnement KOOL

ms 01/15/86 1008.0 fwt Wed

```
; GENERATION AUTOMATIQUE D'ENVIRONNEMENT KOOL
;
; Mise en oeuvre:
; ===============
;
; 1. Appel de l'interpreteur le_lisp:
;      lelisp 8 -r PublicKool
;
; 2. Chargement du programme:
;      (load ms)
;         - le programme charge le fichier des correctifs
;         - il cree une KBase (K) et une Layer (F) de travail
;
; 3. Definir la Super Classe:
;      (mesure)
;      /
;      /
;      i /
;      s_bidon /
;      l /
;
;      m_bidon
;
; 4. Generer les fonctions: (autant de fois qu'on le desire)
;      (genere_fonction)
;      <nombre_de_fonctions>
;
; 5. Generer les classes: (autant de fois qu'on le desire)
;      (genere_classe)
;      <nombre_de_classes>
;      <nombre_d_instances>
;      <nombre_de_cvars>
;      <nombre_d_ivars_1>
;      <nombre_d_ivars_2>
;      <nombre_d_ivars_3>
;      <nombre_d_ivars_4>
;      <nombre_d_ivars_5>
;      <nombre_d_ivars_6>
;
; 6. Generer les regles: (autant de fois qu'on le desire)
;      (genere_regle)
;      <nombre_de_regles>
;      <nombre_de_premisses_sur_cvars>
;      <nombre_de_premisses_sur_ivars>
;      <nombre_de_premisses_sur_ivars_par_hierarchie>
;
;      <nombre_de_premisses_appelant_une_fonction>
;      <nombre_de_conclusions_sur_ivars>
;      <nombre_de_conclusions_creant_une_instance>
;      <nombre_de_conclusions_appelant_la_messagerie>
;      <nombre_de_conclusions_appelant_une_fonction>
;      <code_d_application_de_la_regle>
```

```
; chargement des fonctions KOOL corrigees
(load bugs)


; numero de la prochaine classe
(defvar class_number 1)


; numero de la prochaine cvar
(defvar cvar_number 1)


; numero du prochain ivar de type 1, 2, 3 ou 4
(defvar ivar_a_number 1)




; numero du prochain ivar de type 5 ou 6
(defvar ivar_b_number 1)


; numero de la prochaine regle
(defvar rule_number 1)


; numero de la prochaine fonction
(defvar function_number 1)


; variable globale contenant le code d'application de la regle courante
(defvar code ())


; definition en variable globale des variables de classes
(defvar CLASS_VAR_LIST)


; definition en variable globale des variables d'instances
(defvar IVAR_LIST)



; liste des noms de classes auxquelles appartiennent chacune des cvars
; Exp.: (classe_1 classe_1 classe_3 classe_3 classe_3 classe_4 classe_5)
;       permet de savoir que:
;         la cvar_1 appartient a la classe 1
;                   2                      1
;                   3                      3
;                   4                      3
;                   5                      3
;                   6                      4
;                   7                      5
;
(defvar liste_des_cvars ())
```

```
; liste des noms de classes auxquelles appartiennent chacune des ivars. Indique
; en plus si l'ivar en question est libre (t) ou si elle a deja ete utilisee
; par une regle (()).
; Exp.: ((classe_1 t) (classe_1 t) (classe_1 ()) (classe_2 t) (classe_3 ()))
;       signifie:
;          l'ivar 1 appartient a la classe 1 et n'a pas encore ete utilisee
;                 2                         1
;                 3                         1 et a deja ete utilisee
;                 4                         2 et n'a pas encore ete utilisee


;                 5                         3 et a deja ete utilisee
;
(defvar liste_des_ivars_a ())


; liste des ivars disponible pour les regles actuelles
(defvar ivars_actuels ())


; les memes sous forme de suite
(defvar suite_ivar ())


; premieres regles de declenchement pour une suite de regles
(defvar premier_declenchement ())


; nombre de premisses
(defvar nbre_pre 0)


; nombre de conclusions
(defvar nbre_con 0)



; ensemble des premisses de la derniere regle generee
(defvar PREMISSES ())


; ensemble des conclusions
(defvar CONCLUSIONS ())


; numero de la prochaine iteration (inference)
(defvar n_iter 1)


; nombre maximum d'iterations a effectuer
(defvar max_iter 0)


; definition de la super classe
(de mesure ()
    (DC Super Class)
)
```

```
; methode bidon simulant un appel a la messagerie


(de m_bidon (self)
; ne fait rien
)


; interface utilisateur pour la definition des classes
(de genere_classe ()
    (print "GENERATION DE CLASSES :")
    (print)
    (print "nombre de classes  : ")
    (setq n1 (read))
    (print "nombre d'instances : ")
    (setq n2 (read))
    (print "nombre de c_vars   : ")
    (setq n3 (read))
    (print "nombre d'ivars_1   : ")
    (setq n4 (read))
    (print "nombre d'ivars_2   : ")
    (setq n5 (read))
    (print "nombre d'ivars_3   : ")
    (setq n6 (read))
    (print "nombre d'ivars_4   : ")
    (setq n7 (read))
    (print "nombre d'ivars_5   : ")


    (setq n8 (read))
    (print "nombre d'ivars_6   : ")
    (setq n9 (read))
    (setq l (append (list n1) (list n2) (list n3) (list n4) (list n5)
                    (list n6) (list n7) (list n8) (list n9)))
; appel a la fonction generatrice des classes
    (Class l)
)


; interface utilisateur pour la definition des regles
(de genere_regle ()
    (print '"GENERATION DE REGLES")
    (print)
    (print "nombre de regles")
    (setq n1 (read))
    (print "nombre de premisses sur cvars")
    (setq n2 (read))
    (print "nombre de premisses sur ivars")
    (setq n3 (read))
    (print "nombre de premisses sur ivars par hierarchie")
    (setq n4 (read))
    (print "nombre de premisses appelant une fonction")
    (setq n5 (read))
```

```
    (print)
    (print " de conclusions sur ivars")
    (setq n6 (read))
    (print "nombre de conclusions creant des instances")
    (setq n7 (read))
    (print "nombre de conclusions appelant la messagerie")
    (setq n8 (read))
    (print "nombre de conclusions appelant une fonction")
    (setq n9 (read))
    (print "code d'application des regles: (w)hen-filled, (t)o-fill, (b)oth")
    (setq n10 (read))
    (setq code n10)
; appel a la routine de generation proprement dite
    (Regle n1 n2 n3 n4 n5 n6 n7 n8 n9 n10)
)


; routine d'activation des regles. Elle accepte le nombre maximum d'iterations
; comme parametre. Elle detruit les fonctions qui ont construit l'environnement
; appele le garbage collector, memorise l'etat du systeme (temps CPU deja
; utilise et etat de la memoire) puis cree une instance de chaque classe.
;
(de debut (max_iter)
    (remfn 'mesure)


    (remfn 'genere_classe)
    (remfn 'genere_regle)
    (remfn 'Function)
    (remfn 'mesure_CLASS_VAR_LIST)
    (remfn 'mesure_IVAR_LIST_a)
    (remfn 'mesure_IVAR_LIST_b)
    (remfn 'mesure_INSTANCE)
    (remfn 'premisse_fct)
    (remfn 'conclusion_fct)
    (remfn 'conclusion_inst)
    (remfn 'conclusion_msg)
    (remfn 'Regle)
    (remfn 'regle_1)
    (remfn 'premisse_cvar)
    (remfn 'premisse_ivar)
    (remfn 'construire_premisse)
    (remfn 'construire_conclusion)
    (remfn 'generation_suite_i)
    (remfn 'mesure_increment)
    (remfn 'ensemble_des_ivars)
    (remfn 'combili)
; appel du garbage-collector et demande de liste d'informations
    (setq gc_debut (gc t))
; memorisation du temps CPU deja utilise (depuis le LOGIN)


    (setq t_debut (runtime))
; appel de la fonction de declenchement
    (declenche)
)
```

```
; cree une nouvelle occurence de chaque classe de l'environnement
(de declenche ()
    (<-- Super New)
    (setq m 1)
    (while (< m class_number)
        (<-- (<-- (concat 'Class_name_ m) New) Fill)
        (setq m (1+ m))
    )
)


; fonction appelee a chaque iteration. Verifie que le nombre maximum
; d'iterations n'est pas atteint. S'il ne l'est pas, ne fait rien. Dans le
; cas contraire, memorise l'environnement actuel, affiche l'etat avant et apres
; execution et provoque une erreur pour arreter le mecanisme d'inference.
;
(de that_s_all_folks ()
    (if (<= n_iter max_iter)


        ()
        (setq t_fin (runtime))
        (setq gc_fin (gc t))
        (print t_debut)
        (print t_fin)
        (print "temps :  " (- t_fin t_debut))
        (print gc_debut)
        (print gc_fin)
; appel a une fonction non definie pour arreter le mecanisme
        (fonction_qui_n_existe_pas)
    )
)


; compte le nombre d'inferences
(de compteur ()
    (setq n_iter (1+ n_iter))
)


; interface utilisateur pour la generation des fonctions
(de genere_fonction ()
    (print "GENERATION DE FONCTIONS")
    (print)


    (print "nombre de fonctions a generer")
    (setq nl (read))
    (Function nl)
)


; routine de generation des fonctions. Cree des fonctions du type:
; (de function_<function_number> (n)
;     (while (> n 0)
;         (setq n (1- n))
;     )
;     (progn 0)
```

```
; )
;
; REMARQUE: en le_lisp, les fonctions recursives ne prennent pas de place dans
;           la pile, c'est pourquoi une fonction aussi simple que celle-ci
;           suffit.
;
(de Function (nombre)
    (while (> nombre 0)
        (setq function_name (concat 'function_ function_number))
        (print "creating  " function_name)
        (setq defin '((n) (while (> n 0) (setq n (1- n))) (progn 0)))
        (setq defin (cons function_name defin))


        (eval (cons 'de defin))
        (setq nombre (1- nombre))
        (setq function_number (1+ function_number))
    )
)


; routine de generation des classes
; toutes les classes generees ont comme attribut META: Class et comme
; attribut Super: Super.
;
(de Class (l)
    (setq n1 (car l))
    (setq n2 (cadr l))
    (setq n3 (caddr l))
    (setq n4 (cadddr l))
    (setq n5 (car (cddddr l)))
    (setq n6 (cadr (cddddr l)))
    (setq n7 (caddr (cddddr l)))
    (setq n8 (cadddr (cddddr l)))
    (setq n9 (car (cddddr (cddddr l))))
    (while (> n1 0)
        (setq NAME (concat 'Class_name_ class_number))
        (print "creating  " NAME)


        (setq META 'Class)
        (setq SUPERS (list 'Super))
        (setq premier_cvar_number cvar_number)
        (setq premier_ivar_a_number ivar_a_number)
        (setq premier_ivar_b_number ivar_b_number)
        (mesure_NEW_CLASS1 META NAME SUPERS ())
        (mesure_DEFCLASS1 NAME n3 n4 n5 n6 n7 n8 n9)
        (mesure_INSTANCE NAME n2)
        (setq class_number (1+ class_number))
        (setq n1 (1- n1))
    )
)

; transformation de la fonction NEW_CLASS1 de KOOL pour permettre la creation
; automatique.
(de mesure_NEW_CLASS1 (SELF NAME SUPER_LIST CLASS_LIST)
    (setq CLASS_LIST (copy (GETLV1 $LAYER 'CONTENTS)))
    (cond
```

```
((MEMQ NAME CLASS_LIST)
   (PRINTM "Class" NAME "already exists in layer" $LAYER)
   (terpri)
   nil)
(t (and (NEW_ABSOLUTE_OBJECT1 SELF NAME)
        (cond


              (t (and SUPER_LIST
                      (not (SMAPC$ SUPER_LIST
                                   '(lambda (SP)
                                            (PUTLV1
                                              SP
                                              'SONS
                                              (append
                                                (GETLV1 SP 'SONS)
                                                (list NAME))))))
                      (PUTLV1 NAME 'SUPERS SUPER_LIST))
                 (putprop NAME t 'CLASS)
                 (and (plist $LAYER)
                      (PUTPROP NAME 'LAYER $LAYER)
                      (PUTLV1 $LAYER
                              'CONTENTS
                              (append CLASS_LIST (list NAME)))
                      (SEND1 $LAYER 'Modified () ()))
                 NAME))))))

; transformation de la routine DEFCLASS1 de KOOL pour permettre la generation
; automatique.
(de mesure_DEFCLASS1 (NAME nombre_cvar nombre_ivar_1 nombre_ivar_2
                      nombre_ivar_3 nombre_ivar_4 nombre_ivar_5


                      nombre_ivar_6)
  (setq DESCRIPTION (GET NAME 'OBJECT))
  (setq CLASS_VAR_LIST '())
  (setq IVAR_LIST '())
  (terpri)
  (setq CLASS_VAR_DES (mesure_CLASS_VAR_LIST nombre_cvar)) (terpri)
  (setq IVAR_LIST_1 (mesure_IVAR_LIST_a nombre_ivar_1 1)) (terpri)
  (setq IVAR_LIST_2 (mesure_IVAR_LIST_a nombre_ivar_2 2)) (terpri)
  (setq IVAR_LIST_3 (mesure_IVAR_LIST_a nombre_ivar_3 3)) (terpri)
  (setq IVAR_LIST_4 (mesure_IVAR_LIST_a nombre_ivar_4 4)) (terpri)
  (setq IVAR_LIST_5 (mesure_IVAR_LIST_b nombre_ivar_5 5)) (terpri)
  (setq IVAR_LIST_6 (mesure_IVAR_LIST_b nombre_ivar_6 6)) (terpri)
  (setq IVAR_DES (append IVAR_LIST_1 IVAR_LIST_2 IVAR_LIST_3
                         IVAR_LIST_4 IVAR_LIST_5 IVAR_LIST_6 '()))
  (setq SELECTOR_LIST '())
  (setq METHODS_DES '())
  (nconc DESCRIPTION
         (nconc (and SELECTOR_LIST
                     (list (cons 'SELECTORS (cons 'METHODS METHOD_DES))))
                (nconc (and CLASS_VAR_LIST
                            (list (list 'CVARS CLASS_VAR_LIST)))
                       (nconc (list (list 'INSTANCES nil))
                              (nconc CLASS_VAR_DES
                                     (nconc (and
```

```
                                                    IVAR_LIST
                                                    (list
                                                      (list 'IVARS IVAR_LIST)))
                                                  IVAR_DES))))))
    NAME)


; generation des variables de classes. Les variables de classes generees ont
; le format suivant: class_var_<cvar_number>: <cvar_number>.
(de mesure_CLASS_VAR_LIST (n)
    (prog (res)
          (setq res '())
     loop (if (= n 0)
              (return (nreverse res)))
          (setq nom (concat 'class_var_ cvar_number))
          (print "              " nom)
          (setq liste_des_cvars (append liste_des_cvars (list (concat 'Class_nam
\ce_ class_number))))
          (setq CLASS_VAR_LIST (append CLASS_VAR_LIST (list nom)))
          (setq res (cons (cons nom (or (progn (list cvar_number))
                                        (list ()))) res))
          (setq cvar_number (1+ cvar_number))
          (setq n (1- n))
          (go loop)


    )
)


; creation des variables d'instances de type 1, 2, 3 et 4.
(de mesure_IVAR_LIST_a (n code)
    (prog (res)
          (setq res '())
     loop (if (= n 0)
              (return (nreverse res)))
          (setq nom (concat 'ivar_a_ ivar_a_number))
          (print "              " nom)
          (setq IVAR_LIST (append IVAR_LIST (list nom)))
          (setq liste_des_ivars_a (append liste_des_ivars_a
                   (list (append (list (concat 'Class_name_ class_number))
                                 (list ())))))
          (if (= code 1)
              (setq res (cons (append (list nom) '(101 "TYPE" "NUMBERP") '())
                              res))
          (if (= code 2)
              (setq res (cons (append (list nom) '(101 "TYPE" "NUMBERP" "INIT" 9
\c9) '())
                              res))
          (if (= code 3)


              (setq res (cons (append (list nom) '(101 "TYPE" "NUMBERP" "ASKABLE
\c" t "DEFAULT" 99) '())
                              res))
          (if (= code 4)
              (setq res (cons (append (list nom) '(101 "TYPE" "NUMBERP" "MEANING
\c" "Signification de la variable" "ASKABLE" t "QUESTION" "Introduire la valeur
\cde la variable" "DEFAULT" 99) '())
                              res))
```

```
            ))))
            (setq ivar_a_number (1+ ivar_a_number))
            (setq n (1- n))
            (go loop)
        )
    )


    ; creation des variables d'instances de type 5 et 6.
    (de mesure_IVAR_LIST_b (n code)
        (prog (res)
            (setq res '())
         loop (if (= n 0)
                (return (nreverse res))
            )
            (if (= code 6)


                (if (<= class_number 1)
                    (return (nreverse res))
                )
            )
            (setq nom (concat 'ivar_b_ ivar_b_number))
            (print "                " nom)
            (setq IVAR_LIST (append IVAR_LIST (list nom)))
            (if (= code 5)
                (setq res (cons (append (list nom) '(() "RELATION" t) '()) res)))
            (if (= code 6)
                (setq res (cons (append (list nom) '(() "TYPE") (list (concat 'Cla
    \css_name_ (random 1 (1- class_number)))) '()) res)))
            (setq ivar_b_number (1+ ivar_b_number))
            (setq n (1- n))
            (go loop)
        )
    )


    ; cree n instances de la classe de nom NAME.
    (de mesure_INSTANCE (NAME n)
        (setq i 1)
        (while (<= i n)
            (print "              instance_" i)


            (<-- (<-- NAME New) Fill)
            (setq i (1+ i))
        )
        (print)
    )


    ; cree les premisses appelant une fonction. La fonction appelee est tiree au
    ; hasard.
    (de premisse_fct (n)
        (setq premisse ())
        (if (<= n 0)
            ()
            (setq increment (mesure_increment n 1 (1- function_number)))
```

```
        (setq courant 1)
        (while (> n 0)
            (setq premisse (append premisse
                                   (list (append
                                           (list (append (list (concat 'function_
                                                                       courant))
                                                         (list '30000)))
                                           (list '=)
                                           (list '0)))))
            (setq courant (+ courant increment))


            (if (>= courant function_number)
                (setq courant 1))
            (setq n (1- n))
        )
        (setq PREMISSES (append PREMISSES premisse))
    )
)


; cree les conclusions appelant une fonction. La fonction est choisie au hasard
(de conclusion_fct (n)
    (setq premisse ())
    (if (<= n 0)
        ()
        (setq increment (mesure_increment n 1 (1- function_number)))
        (setq courant 1)
        (while (> n 0)
            (setq premisse (append premisse
                                   (list (append (list (concat 'function_
                                                               courant))
                                                 (list '30000)
                                                 ))))
            (setq courant (+ courant increment))
            (if (> courant function_number)


                (setq courant 1))
            (setq n (1- n))
        )
        (setq CONCLUSIONS (append CONCLUSIONS premisse))
    )
)


; cree les conclusions qui creent des instances d'une classe.
(de conclusion_inst (n)
    (setq conclusion ())
    (while (> n 0)
        (setq conclusion (append conclusion
                                 (list (append '(Make)
                                               (list (concat '*
                                                             (car (nth (random 1 (1- iv
\car_a_number)) liste_des_ivars_a))))))))
        (setq n (1- n))
    )
    (setq CONCLUSIONS (append conclusion CONCLUSIONS))
)
```

```
; cree les conclusions faisant appel a la messagerie


(de conclusion_msg (n)
    (setq conclusion ())
    (while (> n 0)
        (setq conclusion (append conclusion
                                 (list (append '(<--)
                                               (list '(car (<-- Super Instances)
\c))
                                               '(s_bidon)))))
        (setq n (1- n))
    )
    (setq CONCLUSIONS (append conclusion CONCLUSIONS))
)


; redefinition de la fonction KOOL ASK_INST_RULE pour la definition automatique
; de la condition d'application des premisses des regles.
(de ASK_INST_RULE (RULE VAR ATT MODE)
    (if (eq MODE 'when-filled)
        (if (or (eq code 'w) (eq code 'b))
            t
            ())
        (if (or (eq code 't) (eq code 'b))
            t
            ())))


)


; transformation de la fonction KOOL DEFRULE1 pour l'enregistrement des regles
(de mesure_DEFRULE1 (RULE RULE_NAME)
    (setq RULE_NAME (car RULE))
    (PRE_COMPILE RULE ())
    (PRINT_RULE1 RULE_NAME)
    (PRINTM "Is rule" RULE_NAME "well understood ?")
    (cond
       (t
          (cond
             (*FROM_EDIT (COMPILE_FROM_EDIT RULE_NAME ()))
             (t (set RULE_NAME RULE_NAME)
                (COMPILE_IF_THEN3
                    RULE_NAME
                    ()
                    ()
                    ()
                    ()
                    ()
                    ()
                    ()
                    ()
```

```
                    ()
                    ()
                    ()
                    ()))))
        (t (EDIT_RULE RULE_NAME))))


; fonction principale de generation des regles
(de Regle (n_fois n_pre_cvar n_pre_ivar n_pre_hie n_pre_fct
           n_con_ivar n_con_create_inst n_con_msg n_con_fct
           code)
    (if (<= n_pre_ivar 0)
        ()
        (setq liste_ivar (ensemble_des_ivars n_fois n_pre_ivar n_con_ivar))
        (setq suite_ivar (generation_suite_i n_pre_ivar n_con_ivar liste_ivar))
    )
    (while (> n_fois 0)
        (print "creating_rule")
        (setq g_regle (append (list (concat 'Rule_ rule_number))
                      (regle_1 n_fois n_pre_cvar n_pre_ivar n_pre_hie n_pre_fct
                               n_con_ivar n_con_create_inst n_con_msg n_con_fct
                               code)))
        (apply 'DEFRULE g_regle)
        (setq rule_number (1+ rule_number))


        (setq n_fois (1- n_fois))
    )
)


; genere une regle
(de regle_1 (nb_fois n_pre_cvar n_pre_ivar n_pre_hie n_pre_fct
             n_con_ivar n_con_create_inst n_con_msg n_con_fct
             code)
    (setq PREMISSES ())
    (setq CONCLUSIONS ())
    (premisse_cvar n_pre_cvar)
    (premisse_ivar nb_fois n_pre_ivar n_con_ivar)
    (premisse_hie n_pre_hie)
    (premisse_fct n_pre_fct)
    (conclusion_inst n_con_create_inst)
    (conclusion_msg n_con_msg)
    (conclusion_fct n_con_fct)
    (setq CONCLUSIONS (append CONCLUSIONS '((compteur))))
;    (if (= nb_fois 1)
;        (setq CONCLUSIONS (append CONCLUSIONS '((declenche))))
;    )
    (progn (append '(IF) PREMISSES '(THEN) CONCLUSIONS))
)




; cree les premisses portant sur les cvars.
(de premisse_cvar (n_pre_cvar)
    (setq res ())
    (if (<= n_pre_cvar 0)
```

```
        ()
        (setq l n_pre_cvar)
        (while (> l 0)
            (setq r (random l (1- cvar_number)))
            (setq res (append res (list (append (list (concat '*
                                                      (nth (1- r) liste_des_cva
\crs)
                                                      '":class_var_"
                                                      r))
                                                '(GT)
                                                '(0)))))
            (setq l (1- l))
        )
    )
    (setq PREMISSES (append res PREMISSES))
)



; cree les premisses et les conclusions portant sur les ivars.
(de premisse_ivar (nb_fois n_pre_ivar n_con_ivar)
    (setq res ())
    (if (<= n_pre_ivar 0)
        ()
        (setq l (length (car suite_ivar)))
        (setq nbre_pre 0)
        (setq nbre_con 0)
        (setq i 1)
        (if (= nb_fois 1)
            (setq suite_ivar (append (list (car suite_ivar))
                                     premier_declenchement
                                     (cdr liste_ivar)))
        )
        (while (> l 0)
            (setq PREMISSES (append (construire_premisse (nth (1- l) (car suite_
\civar))) PREMISSES))
            (if (<> (nth (1- l) (car suite_ivar)) (nth (1- l) (cadr suite_ivar))
\c)
                (setq CONCLUSIONS (append (construire_conclusion (nth (1- l) (ca
\cdr suite_ivar))) CONCLUSIONS))
            )
            (if (<> (abs (nth (1- l) (car suite_ivar))) (abs (nth (1- l) (cadr s
\cuite_ivar))))


                (setq PREMISSES (append (construire_premisse (nth (1- l) (cadr s
\cuite_ivar))) PREMISSES))
            )
            (setq l (1- l))
        )
        (if (>= nbre_pre n_pre_ivar)
            ()
            (setq i 1)
            (while (< nbre_pre n_pre_ivar)
                (setq PREMISSES (append (construire_premisse (nth (1- i) (car (c
\cdr suite_ivar)))) PREMISSES))
                (setq i (1+ i))
                (if (> i (length (car suite_ivar)))
                    (setq i 1)
```

```
                )
            )
        )
        (if (>= nbre_con n_con_ivar)
            ()
            (setq i 1)
            (while (< nbre_con n_con_ivar)
                (setq CONCLUSIONS (append (construire_conclusion (nth (1- i) (ca
\cr (cdr suite_ivar)))) CONCLUSIONS))
                (setq i (1+ i))


                (if (> i (length (car suite_ivar)))
                    (setq i 1)
                )
            )
        )
        (setq suite_ivar (cdr suite_ivar))
        (if (not (null (cdr suite_ivar)))
            ()
            (setq liste_ivar (ensemble_des_ivars nb_fois n_pre_ivar n_con_ivar))
            (setq suite_ivar (generation_suite_i n_pre_ivar n_con_ivar liste_iva
\cr))
        )
    )
)


; cree les premisses faisant appels a la hierarchie.
(de premisse_hie (p)
    (setq res ())
    (while (> p 0)
        (setq r (random 1 (1- ivar_a_number)))
        (setq res (append res (list (append (list (concat '*
                                                  (car (nth (1- r) liste
\c_des_ivars_a))


                                                  '":i"))
                                           '(GT)
                                           '(0)))))
        (setq p (1- p))
    )
    (setq PREMISSES (append PREMISSES res))
)


; construit un premisse portant sur l'ivar p
(de construire_premisse (p)
    (setq pre (list (concat '*
                            (car (nth (1- (abs p)) liste_des_ivars_a))
                            '":ivar_a_"
                            (abs p))))
    (if (> p 0)
        (setq pre (append pre '(GT)))
        (setq pre (append pre '(LT)))
    )
    (setq pre (list (append pre '(100))))
    (setq nbre_pre (1+ nbre_pre))
    (progn pre)
)
```

```
; construit une conclusion portant sur l'ivar p.
(de construire_conclusion (p)
    (setq con (list (concat '*
                            (car (nth (1- (abs p)) liste_des_ivars_a))
                            '":ivar_a_"
                            (abs p))))
    (setq con (append con '(<-)))
    (if (> p 0)
        (setq con (list (append con
                                (list (append (list '+)
                                              (list (concat '*
                                                            (car (nth (1- (abs p
\c)) liste_des_ivars_a))
                                                            '":ivar_a_"
                                                            (abs p)))
                                              (list '2))))))
        (setq con (list (append con '(99))))
    )
    (setq nbre_con (1+ nbre_con))
    (progn con)
)



(de generation_suite_i (n_pre n_con liste_ivar)
    (setq actuel (car liste_ivar))
    (if (>= n_pre n_con)
        (setq nombre n_con)
        (setq nombre n_pre)
    )
    (setq index 1)
    (setq longueur_suite 1)
    (setq suite (list (car liste_ivar)))
    (setq possible t)
    (setq nbr_fois nombre)
    (setq premier_declenchement suite)
    (while (and (> nbr_fois 0) possible)
        (setq modification ())
        (setq anc_actuel actuel)
        (setq anc_index index)
        (setq fois (length actuel))
        (while (and (> fois 0) possible)
            (setq actuel (niemet (1- index) (- (nth (1- index) actuel)) actuel))
            (if (member actuel suite)
                (setq actuel (niemet (1- index) (- (nth (1- index) actuel)) actu
\cel))
                (setq modificaction t)
                (setq fois (1- fois))


            )
            (setq index (1+ index))
            (if (> index (length actuel))
                (setq index 1)
            )
            (if (= index anc_index)
                (ifn modification
```

```
                    (setq possible ())
                )
                (setq fois 0)
            )
        )
        (ifn possible
            (if (null (cadr liste_ivar))
                ()
                (setq suite (append suite (list (cadr liste_ivar))))
            )
            (if (null actuel)
                ()
                (setq suite (append suite (list actuel)))
                (setq longueur_suite (1+ longueur_suite))
                (setq nbr_fois (1- nbr_fois))
            )
        )


    )
    (progn suite)
)


(de mesure_increment (n min max)
    (setq increment (div (- max min) n))
    (if (= increment 0)
        (setq increment 1)
    )
    (progn increment)
)


(de ensemble_des_ivars (nb_fois n_pre_ivar n_con_ivar)
    (if (>= n_pre_ivar n_con_ivar)
        (setq nombre n_con_ivar)
        (setq nombre n_pre_ivar)
    )
    (setq n nb_fois)
    (setq increment (div ivar_a_number n))
    (setq actuel 1)
    (setq tour 0)
    (setq res ())


    (setq suite ())
    (setq possible t)
    (while (and possible (> n 0))
        (setq l ())
        (setq nl nombre)
        (while (and possible (> nl 0))
            (if (eq (cadr (nth (1- actuel) liste_des_ivars_a)) ())
                ()
                (while (and possible (eq (cadr (nth (1- actuel) liste_des_ivars_
\ca)) t))
                    (setq actuel (1+ actuel))
                    (ifn (> actuel (1- ivar_a_number))
                        ()
                        (setq actuel 1)
```

```
                        (setq tour (1+ tour))
                        (if (= tour 2)
                            (setq possible ())
                        )
                    )
                )
            )
            (ifn (eq (cadr (nth (1- actuel) liste_des_ivars_a)) ())
                 ()
                 (setq l (append (list actuel) l))


                 (setq liste_des_ivars_a (niemet (1- actuel) (append (list (car
\c(nth (1- actuel) liste_des_ivars_a))) (list t)) liste_des_ivars_a))
                 (setq actuel (+ actuel increment))
                 (if (> actuel (1- ivar_a_number))
                     (setq actuel 1)
                 )
            )
            (setq nl (1- nl))
        )
        (if (and possible (not (null l)))
            (setq res (append (list l) res))
        )
        (setq n (1- n))
    )
    (progn res)
)


(de combili (nombre nb_fois)
    (setq res (* nombre nb_fois))
    (progn res)
)



; remplace l'element n de la liste liste par l'element newn. L'element pouvant
; etre lui-meme une liste.
(de niemet (n newn liste)
    (setq n (1+ n))
    (if (<= n 1)
        (setq tete ())
        (setq tete (firstn (1- n) liste))
    )
    (if (>= n (length liste))
        (setq queue ())
        (setq queue (lastn (- (length liste) n) liste))
    )
    (progn (append tete (list newn) queue))
)


; modification de la fonction KOOL USR pour compter le nombre d'inferences.
(de USR (RULE_NAME TRIGG FORM_NUMBER)
    (that_s_all_folks)
;    (printstack 1)
```

```
    (let ((ANTE_MODIF (REMFLAG RULE_NAME 'MODIFIED))
          (CLAUSE_NUMBER (GET_CLAUSE_NUM RULE_NAME))
          (USED_VARS (GET_USED_VARS RULE_NAME))
          (VAR_NAMES (GET_VAR_NAMES RULE_NAME))


          (VAR_LIST (GET_VAR_LIST RULE_NAME FORM_NUMBER))
          (MODIFIED_INSTANCES ()))
         (putprop RULE_NAME (1+ (get RULE_NAME 'ACTIVE)) 'ACTIVE)
         (letv (cons '*RULE_FIRED* USED_VARS)
               (makelist (1+ (length USED_VARS)) '())
               (set TRIGG SELF)
               (mapc (lambda ((v . w))
                             (putprop v (GETLV1 w 'INSTANCES) 'INSTANCES))
                     VAR_NAMES)
               (EVAL_RULE
                   (GET_PREM_EXPR RULE_NAME FORM_NUMBER)
                   (GET_CONCL_EXPR RULE_NAME))
               (when (or ANTE_MODIF (GET_MODIF_OBJECTS RULE_NAME))
                     (FLAG RULE_NAME 'MODIFIED))
               (putprop RULE_NAME (1- (get RULE_NAME 'ACTIVE)) 'ACTIVE))))



(de ASK_FOR_SAVER (RULE_NAME ANSWER EXIT_LOOP)
    (PRINTM "on which file do you want to save this rule ?")
    (let ((#:prog:i t)
          (#:system:body
                '((setq)


                  (go #:do:end)
                  #:do:start
                  (if (and EXIT_LOOP) (return (progn 'bidon)))
                  (setq ANSWER 'bidon)
                  (if (member ANSWER (FETCH_RULE_FILES))
                      (return (progn (<-- ANSWER 'Modified) ANSWER)))
                  (cond
                     ((eq ANSWER '?)
                        (PRINTM "you can choose among :")
                        ($MAPC$ (FETCH_RULE_FILES)
                                '(lambda (FILE) (prin FILE) (prin ",")))
                        (terpri)
                        (PRINTM "or define a new one :"))
                     (t (terpri)
                        (PRINTM "new file :" ANSWER "?")
                        (cond
                           (t
                              (<-- RuleLayer 'New ANSWER)
                              (<-- ANSWER 'Modified)
                              (setq EXIT_LOOP t))
                           (t (PRINTM "which file then :")))))
                  (setq)
                  #:do:end
                  (if (null t) (return (progn nil)) (go #:do:start))))
```

```
        (#:system:res ())
        (#:prog:pc ()))
       (setq #:prog:pc #:system:body)
       (tag return
            (while #:prog:pc
                   (tag go
                        (if (symbolp (car #:prog:pc))
                            (nextl #:prog:pc)
                            (setq #:system:res (eval (nextl #:prog:pc)))
                        )))
            #:system:res)))

; creation de la KBase et de la Layer de travail.
(<-- KBase New 'K)
(<-- K)

(<-- Layer New 'L)
(<-- L)



r 10:37 1.422 109 level 5
```

ANNEXE 2 : Les programmes de tests en Prolog

1) Inversion de listes

```
test(X) :-
   core,
   loop(1,X,1).

loop(V,U,I) :-
   V >= U, !,
   prgm,
   core.

loop(V,U,I) :-
   prgm,
   N is V + I,
   loop(N,U,I).

prgm :- nreverse([1,2,3,4,5,6,7,8,9,10,11,12,13,15,16,17,18,19,20,
                  21,22,23,24,25,26,27,28,29,30],X).

nreverse([X|L0],L) :- nreverse(L0,L1) , concatenate(L1,[X],L).
nreverse([],[]).

concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
concatenate([],L,L).
```

2) Le "quick-sort"

```
test(X) :-
   core,
   loop(1,X,1).

loop(V,U,I) :-
   V >= U, !,
   prgm,
   core.

loop(V,U,I) :-
   prgm,
   N is V + I,
   loop(N,U,I).

prgm :- qsort([27,74,17,33,94,18,46,83,65, 2,
               32,53,28,85,99,47,28,82, 6,11,
               55,29,39,81,90,37,10, 0,66,51,
                7,21,85,27,31,63,75, 4,95,99,
               11,28,61,74,18,92,40,53,59, 8],X,[]).

qsort([X|L],R,R0) :-
   partition(L,X,L1,L2),
   qsort(L2,R1,R0),
   qsort(L1,R,[X|R1]).

qsort([],R,R).

partition([X|L],Y,[X|L1],L2) :- X =< Y, !, partition(L,Y,L1,L2).

partition([X|L],Y,L1,[X|L2]) :- partition(L,Y,L1,L2).

partition([],_,[],[]).
```

3) Transposition d'une liste de symboles en une liste de nombres

```
test(X) :-
   core,
   loop(1,X,1).

loop(V,U,I) :-
   V >= U, !,
   prgm,
   core.

loop(V,U,I) :-
   prgm,
   N is V + I,
   loop(N,U,I).

prgm :- serialise("ABLE WAS I ERE I SAW ELBA",X).

serialise(L,R) :- pairlists(L,R,A), arrange(A,T), numbered(T,1,N).

pairlists([X|L],[Y|R],[pair(X,Y)|A]) :- pairlists(L,R,A).
pairlists([],[],[]).

arrange([X|L],tree(T1,X,T2)) :- split(L,X,L1,L2), arrange(L1,T1),
  arrange(L2,T2).
arrange([],void).

split([X|L],X,L1,L2) :- !, split(L,X,L1,L2).
split([X|L],Y,[X|L1],L2) :- before(X,Y), !, split(L,Y,L1,L2).
split([X|L],Y,L1,[X|L2]) :- before(Y,X), !, split(L,Y,L1,L2).
split([],_,[],[]).

before(pair(X1,Y1),pair(X2,Y2)) :- X1 < X2.

numbered(tree(T1,pair(X,N1),T2),N0,N) :-
  numbered(T1,N0,N1), N2 is N1+1, numbered(T2,N2,N).
numbered(void,N,N).
```

4) Interrogation d'une base de données

```
test(X) :-
   core,
   loop(1,X,1).

loop(V,U,I) :-
   V >= U, !,
   prgm,
   core.

loop(V,U,I) :-
   prgm,
   N is V + I,
   loop(N,U,I).

prgm :- query([C1,D1,C2,D2]),fail.

prgm.

query([C1,D1,C2,D2]) :- density(C1,D1), density(C2,D2), D1 > D2,
   20 * D1 < 21 * D2.

density(C,D) :- pop(C,P), area(C,A), D is (P * 100.0) / A.

pop(china,        8250).
pop(india,        5863).
pop(ussr,         2521).
pop(usa,          2119).
pop(indonesia,    1276).
pop(japan,        1097).
pop(brazil,       1042).
pop(bangladesh,    750).
pop(pakistan,      682).
pop(w_germany,     620).
pop(nigeria,       613).
pop(mexico,        581).
pop(uk,            559).
pop(italy,         554).
pop(france,        525).
pop(philippines,   415).
pop(thailand,      410).
pop(turkey,        383).
pop(egypt,         364).
pop(spain,         352).
pop(poland,        337).
pop(s_korea,       335).
pop(iran,          320).
pop(ethiopia,      272).
pop(argentina,     251).
```

```
area(china,        3380).
area(india,        1139).
area(ussr,         8708).
area(usa,          3609).
area(indonesia,     570).
area(japan,         148).
area(brazil,       3288).
area(bangladesh,     55).
area(pakistan,      311).
area(w_germany,      96).
area(nigeria,       373).
area(mexico,        764).
area(uk,             86).
area(italy,         116).
area(france,        213).
area(philippines,    90).
area(thailand,      200).
area(turkey,        296).
area(egypt,         386).
area(spain,         190).
area(poland,        121).
area(s_korea,        37).
area(iran,          628).
area(ethiopia,      350).
area(argentina,    1080).
```

5) Gestion dynamique d'une base de données

```
test(X) :-
   core,
   loop(1,X,1).

loop(V,U,I) :-
   V >= U, !,
   prgm,
   core.

loop(V,U,I) :-
   prgm,
   N is V + I,
   loop(N,U,I).

prgm :- test1(100).
prgm.

test2(X,X).
test2(X,Y) :- fact(X), Z is X + 1, assert(fact(Z)), test2(Z,Y).

test1(Y) :- assert(fact(1)), test2(1,Y).
```

6) Le Sieve Benchmark

```
test(X) :-
   core,
   loop(1,X,1).

loop(V,U,I) :-
   V >= U, !,
   prgm,
   core.

loop(V,U,I) :-
   prgm,
   N is V + I,
   loop(N,U,I).

prgm :- sieve(10).

   sieve(LIMIT) :-
      loop1(0,LIMIT,1),
      loop2(0,LIMIT,1).

   loop1(Value,Upper_Limit,Increment) :-
      Value >= Upper_Limit, !,
      assert(flags(Value)).
   loop1(Value,Upper_Limit,Increment) :-
      assert(flags(Value)),
      New_Value is Value + Increment,
      loop1(New_Value,Upper_Limit,Increment).

   loop2(Value,Upper_Limit,Increment) :-
      Value >= Upper_Limit, !,
      s(Value,Upper_Limit,Increment).
   loop2(Value,Upper_Limit,Increment) :-
      s(Value,Upper_Limit,Increment),
      New_Value is Value + Increment,
      loop2(New_Value,Upper_Limit,Increment).

   s(I,LIMIT,Increment) :-
      flags(I),
      Prime is I + I + 3,
      K is I + Prime,
      loop3(K,LIMIT,Prime).

   s(I,LIMIT,Increment).
```

```
loop3(Value,Upper_Limit,Increment) :-
   Value > Upper_Limit, !.
loop3(Value,Upper_Limit,Increment) :-
   flags(Value),
   retract(flags(Value)),
   New_Value is Value + Increment,
   loop3(New_Value,Upper_Limit,Increment).
loop3(Value,Upper_Limit,Increment) :-
   New_Value is Value + Increment,
   loop3(New_Value,Upper_Limit,Increment).
```

7) Le Float benchmark

```
test(X) :-
   core,
   loop(1,X,1).

loop(V,U,I) :-
   V >= U, !,
   prgm,
   core.

loop(V,U,I) :-
   prgm,
   N is V + I,
   loop(N,U,I).

prgm :- sieve(10.0).

   sieve(LIMIT) :-
      loop1(0.0,LIMIT,1.0),
      loop2(0.0,LIMIT,1.0).

   loop1(Value,Upper_Limit,Increment) :-
      Value >= Upper_Limit, !,
      assert(flags(Value)).
   loop1(Value,Upper_Limit,Increment) :-
      assert(flags(Value)),
      New_Value is Value + Increment,
      loop1(New_Value,Upper_Limit,Increment).

   loop2(Value,Upper_Limit,Increment) :-
      Value >= Upper_Limit, !,
      s(Value,Upper_Limit,Increment).
   loop2(Value,Upper_Limit,Increment) :-
      s(Value,Upper_Limit,Increment),
      New_Value is Value + Increment,
      loop2(New_Value,Upper_Limit,Increment).

   s(I,LIMIT,Increment) :-
      flags(I),
      Prime is I + I + 3.0,
      K is I + Prime,
      loop3(K,LIMIT,Prime).

   s(I,LIMIT,Increment).
```

```
loop3(Value,Upper_Limit,Increment) :-
   Value > Upper_Limit, !.
loop3(Value,Upper_Limit,Increment) :-
   flags(Value),
   retract(flags(Value)),
   New_Value is Value + Increment,
   loop3(New_Value,Upper_Limit,Increment).
loop3(Value,Upper_Limit,Increment) :-
   New_Value is Value + Increment,
   loop3(New_Value,Upper_Limit,Increment).
```